Imperial College London
Department of Computing

MEng Individual Project

MCMAS-Dynamic

Symbolic Model Checking for Linear Dynamic Logic and Several Temporal and Epistemic Extensions

---

Author:
Jeremy Liang An Kong

Supervisor:
Prof. Alessio Lomuscio

Second Marker:
Dr. Krysia Broda

13 June 2016

Abstract

Linear Dynamic Logic (LDL) is a relatively modern specification language, put forth by Vardi in 2011. Specifying properties in LDL is attractive owing to its high expressivity (equivalent to ω-regular languages and monadic second order logic) and clear and simple syntax. Furthermore, verifying LDL properties is computationally attractive as the problem has a reasonable (PSPACE) complexity. This report documents the development of novel, practical algorithms for LDL model checking. We extend LDL with epistemic modalities (giving rise to LDLK), with full branching time semantics (CDL*K), and with finite trace semantics (CDL*$_f$K). We show that these extensions preserve the PSPACE-completeness of LDL model checking, and develop algorithms for model checking these languages.

We also document the design and implementation of these algorithms in MCMAS-Dynamic, which is an extension of the MCMAS model checker. MCMAS-Dynamic is the first model checker that supports LDL, LDLK, CDL*K and CDL*$_f$K properties. We comprehensively evaluate our tool, considering its correctness and scalability over several practical scenarios and specifications. We demonstrate our tool's resilience to large models, its high expressive power, as well as its limitations concerning large formulae. Our results suggest that in many cases, LDL and even CDL*K model checking can be feasible in practice.

Acknowledgements

I would like to thank:

- Prof. Alessio Lomuscio, for his guidance and support throughout the entire project. His suggestions and feedback have certainly spurred much exploration, and helped me develop a substantially more expressive and powerful tool than I initially thought possible.
- Dr. Krysia Broda, for her input, feedback and suggestions especially during the early stages of the project.
- Prof. Michael Huth, my personal tutor, for his support and advice over the years.
- Bryan, James, Tom and Andrea, with whom I've had opportunities to discuss and validate some of my ideas (even though they may not initially have been as acquainted with systems verification).
- my family and friends for their support throughout my four years at Imperial.

Contents

Chapter 1

Introduction

Computers and computer systems have been becoming increasingly more integral in our daily lives. We rely on their correctness in a variety of circumstances – clear examples include smartphones, point-of-sale systems and electronic mail. However, the development of these systems (both in terms of hardware and software) is a human process and thus naturally subject to error. Developers often do make significant efforts in ensuring that their systems are robust (through writing tests, code reviews and, in certain cases, static analysis tools such as FindBugs [4]). These processes have shown promise in reducing defect rates [15, 56, 12], but they are fundamentally *incomplete* – they can demonstrate that bugs and errors exist, but have no way of showing that a system is wholly correct. In view of the potentially catastrophic consequences of hardware and software errors, such as a race condition in the Therac-25 radiotherapy machine leading to massive radiation overdoses and the death of several patients [61], or a floating point division bug in Intel's Pentium processor leading to a recall costing the company an estimated $475 million [74], in many cases it is prudent to verify the correctness of computer systems.

Formal verification of both hardware and software has seen increasing take-up in industry, especially in safety-critical applications. These techniques can prove that programs (or program fragments) satisfy a given specification (modulo bugs or errors in implementation of said technique). For example, various static analysis and model checking tools were used in the design and implementation of flight control software for the *Curiosity* rover that landed on Mars – and these tools found many bugs, even leading to a redesign of one of the subsystems in the flight control software [51]. Other industrial applications of model checking include verifying correctness of microprocessors at Intel [47], the world's largest semiconductor vendor by market share [44], and checking that third-party device drivers use the Windows driver API correctly [14].

Model checking a computer system involves building a model $\mathcal{M}$ encompassing the possible states the system can be in as well as transitions between these states, and checking that some specification ϕ holds across (typically) *all* reachable states of the model. However, computer systems in practice tend to be very large, and these models tend to grow exponentially as more components are added – this is known as the *state explosion problem* [69]. Thus, for the technique to scale effectively we cannot simply explicitly build the model; we need to construct and verify properties in a more sophisticated manner. This is covered in greater detail in Section 2.4.

Multi-agent systems refer to computer systems involving interactions between multiple intelligent agents. These agents act in different ways based on the external environment that they can observe – an *ideal rational agent* is one that maximises its performance based on the ob-

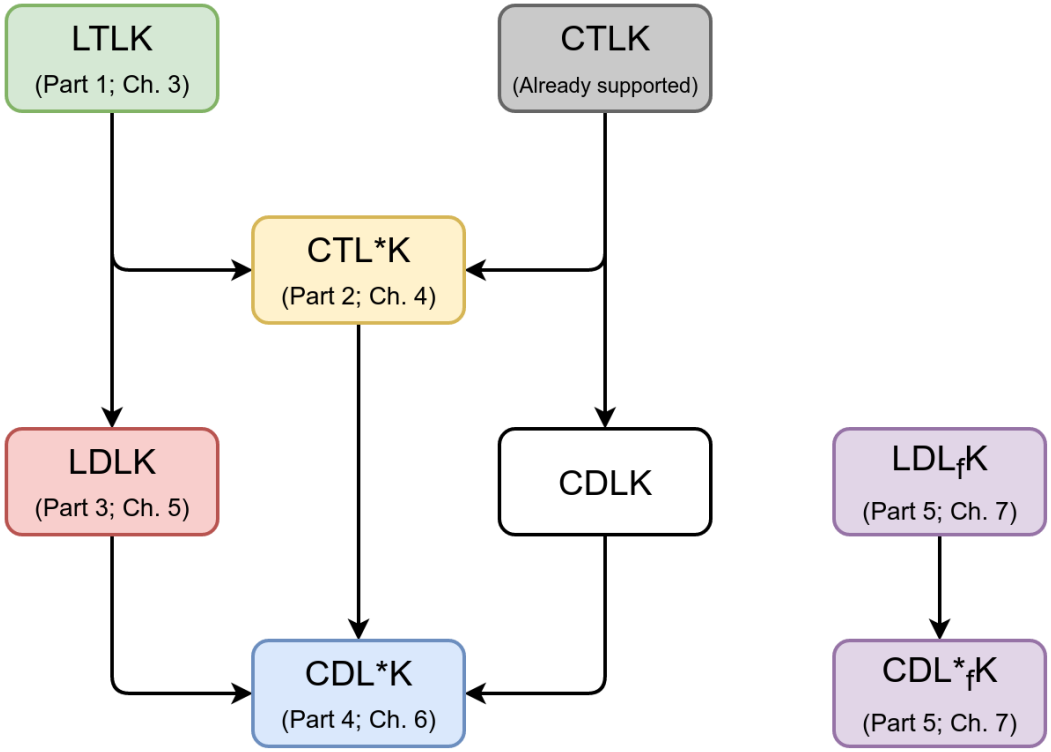


Figure 1.1: Illustration of relationships between various temporal-epistemic logics. The arrows denote that one logic is strictly subsumed by another: $A \to B$ if every property expressible in A is also expressible in B, and there exists a property expressible in B not expressible in A.

servations it makes [82]. These systems see potential applications across a diverse spectrum of fields, such as reasoning about auctions [77], traffic control [25] and robotic swarms [58].

Specifications for model checking are often expressed in temporal logic [79]; these logics vary in their expressive power as well as complexity for model checking. There are several tools available that support model checking of temporal logic properties in multi-agent systems. Examples include MCMAS, which supports Computation Tree Logic (CTL) and Alternating-Time Temporal Logic (ATL) with epistemic and deontic modalities [65], and MCK which supports LTL, CTL and CTL* with epistemic modalities [5]. Over the course of this project, we implemented a model checker for several temporal logics of increasing expressivity in the context of multi-agent systems, as illustrated in Figure 1.1.

It is well-known that CTL and LTL have "incomparable" expressive power [68]; there are properties that can be expressed in LTL but not in CTL and vice versa. However, there exist claims that LTL is frequently considered better suited to writing specifications for systems occurring in practice [79, 95]. This project will, as a first step, focus on implementing an open source model-checker for LTLK in the context of multi-agent systems; the aforementioned MCK is closed-source. While algorithms for model checking LTL are well-known [28, 79], extending these to handle epistemic modalities is less so. We then extend the model checker to handle full branching time logic with epistemic modalities (CTL*K) in multi agent systems.

We then turn our attention to Linear Dynamic Logic (LDL), an extension of LTL with dy-

namic modalities expressed in syntax analogous to Propositional Dynamic Logic [35]. LDL is attractive as a specification language as it is strictly more expressive than LTL. In addition to LTL properties, it also allows other structural properties to be expressed as well, such as a formula ϕ holding in *even* states on every path (impossible in LTL [35] without formulae of infinite size, but succinctly $[(\top;\top)^*]\phi$ in LDL). It also has incomparable expressive power with CTL*; there are statements expressible in each that are not expressible in the other (in particular, the 'even states' property is also not expressible in CTL*). Furthermore, such regular properties have seen use in industrial applications [47]. However, LDL model checking maintains the same worst-case complexity (like LTL and CTL* model checking, it is PSPACE-complete [96]). We seek to implement a model checker for LDL with epistemic modalities (LDLK); we are not currently aware of any existing model checkers that support LDL or LDLK specifications (open source or otherwise).

We then proceed to generalise the LDLK algorithm to support full branching time dynamic logic with epistemic modalities (CDL*K). This subsumes both LDLK and CTL*K, allowing us to express complex structural properties, such as "in every even state on every path, it is possible for ϕ to hold after an odd number more steps" $(A[(\top;\top)^*]E\langle\top;(\top;\top)^*\rangle\phi)$ in a succinct way. We also implement counterexample generation for CDL*K properties. Furthermore, we demonstrate that model checking CDL*K also maintains the PSPACE-completeness of LTLK, CTL*K and LDLK model checking, in spite of its strictly greater expressiveness. We also explore possibilities for verifying CDL*K properties over finite traces, show that the CDL*K model checking problem over finite traces is PSPACE-complete and provide a reduction of this problem to the infinite trace CDL*K model checking problem. We are not aware of any existing model checkers that support CDL* or CDL*K specifications (on finite or infinite traces).

1.1 Primary Objectives

The goal of this project is to implement model checkers for LTL, CTL* and LDL and their epistemic extensions in the context of multi-agent systems. We also aim to implement a model checker for the full branching time extension of LDL with epistemic modalities, CDL*K, as well as a model checker for LDL and CDL*K over finite traces. At a high level, we seek to achieve this by carrying out the following for *each* specification language:

1. **Research existing methods for verifying formulae expressed in the relevant specification language.** For LTL and CTL*, we need to find out more information about existing algorithms and evaluate whether they are amenable to symbolic model checking (such as the *tableau construction* algorithm [28] or the recursive approach of [39]), as well as whether and how these algorithms can be adapted to handle epistemic modalities.

   For LDL and its extensions, this is more difficult owing to its relative novelty; there does not appear to be a "standard" algorithm for LDL model checking.

2. **Design a suitable algorithm for verifying properties specified in the relevant specification language.** There appear to be many possible choices for LTL and CTL*, though due to the aforementioned state explosion problem it is important to represent the state space efficiently. Furthermore, a method of handling the epistemic modalities also needs to be devised.

   For LDL there does not appear to be an existing algorithm (the closest we have found

appears to be the alternating automaton construction presented in [41], though it features a quantification over potentially arbitrarily many paths). As before, we also need to determine how to incorporate epistemic modalities in our algorithm. Our algorithm will thus need to be original, and hence justifying its correctness and complexity is also important. This applies to the extensions of LDL we plan to explore as well, which to the best of our knowledge are novel.

3. **Implement a tool allowing appropriate specifications to be verified over multi-agent systems.** We plan on implementing this as an extension of an existing model checker – otherwise, we would need to spend time developing many other components (such as a specification language, parser, tool for determining fair and reachable states *etc.*) that are not intended to be the focus of this project.

   The tool should be able to, given some specification of a multi-agent system and temporal-epistemic properties, determine if said properties are satisfied. In particular, it should support fairness constraints, as well as the generation of suitable counterexample or witness traces as appropriate.

4. **Evaluate the tool's correctness, performance and scalability.** We will verify the tool's correctness by manually devising many test cases designed to exercise the tool's functionality, as well as comparing its behaviour with equivalent specifications in languages that MCMAS already supports.

   The tool will also be evaluated on its scalability, using several well-known problems in the literature such as the *dining cryptographers*. Although the model checking problems for these languages are PSPACE-complete (for LTL, CTL* and LDL this is established in [39] and [96]; we will prove this for the extensions), encouraging empirical results for LTL, at least, have been obtained in practice if suitable data structures are used [28].

1.2 Challenges

We faced several challenging tasks throughout this project; in particular, we had to translate complex, abstract algorithmic concepts into (relatively) low-level C++ code with no prior experience in this field beyond the Systems Verification and Software Reliability courses. Some of the primary challenges we faced were as follows:

1. **Lack of (documented) existing algorithms.** While algorithms for model-checking LTL and CTL* are well-known, how these may be extended with epistemic modalities and multi-agent systems appears considerably less well-known (that said, they certainly *exist*, since MCK is able to check such properties). We applied ideas from [70], [40] and [28] in developing algorithms for model-checking LTLK and CTL*K that, to the best of our knowledge, are not explicitly documented elsewhere. Some adaptations to the aforementioned algorithms were also necessary during implementation, to deal with MCMAS' existing encoding of the model.

   Furthermore, we have not found any algorithms for model checking LDL or CDL* (with or without epistemic modalities), and there are no existing model checkers for it. There is an alternating automaton construction for LDL specifications [41], which does indeed confirm that LDL model checking is feasible, though there are some obstacles to an efficient concrete implementation (in particular, the construction involves a quantification over a

potentially unbounded number of ϵ-paths). In general, we find that the lack of available information made it considerably more difficult to make inroads into the LDL and CDL* model checking problems.

2. **Working with the legacy MCMAS codebase.** The existing MCMAS codebase while open-source appears to have accrued substantial technical debt, with considerable use of 'God objects' with many responsibilities (in particular, the `bdd_parameters` struct) and global variables (such as `agents`, a vector of agent specifications). This did lead to confusing bugs due to undocumented and, in some cases, unexpected dependencies between methods[1]. Furthermore, while user documentation for MCMAS is available, *technical* documentation was nonexistent. We were also unable to find any existing tests (unit, regression or otherwise) for MCMAS at all, which initially made it difficult to ascertain if our changes had wider-reaching effects.

   Furthermore, MCMAS itself did have subtle, complex bugs that were difficult to identify (and this was exacerbated by the aforementioned technical debt). We discovered (and fixed) a bug in MCMAS concerning witness generation for CTL formulas of the form $EG\phi$.

3. **High computational complexity.** Model checking of LTL and CTL* is PSPACE-complete, and the adaptations we have made to support epistemic modalities preserve this (intuitively, these involve polynomially many calls to a plain LTL model checking algorithm). When faced with poor performance, it was at times difficult to determine whether said poor performance was a fundamental limitation of the algorithms being used or whether the implementation of the algorithms was inefficient (as far as we know, both situations have arisen in practice).

   Model checking of LDL is also PSPACE-complete [96]; we prove that this also holds for CDL* (Section 6.3), and as before our adaptations to support epistemic modalities preserve this. This also holds for the finite trace versions (for LDL, this was shown in [35]; for CDL* we show it in Section 7.2). When testing for *correctness*, we also encountered difficulties verifying that certain edge cases were respected (because our attempted tests involved constructing very specific formulae that would trigger said edge case, and these were, at times, too large to feasibly verify).

1.3 Contributions

The main contributions of our project are as follows:

1. **Open source model checking tool for LTLK and CTL*K.** We extended the functionality of MCMAS, a leading model checker for multi-agent systems [10], to verify properties specified in LTLK and CTL*K. We also implemented support for counterexample and witness generation. This contribution would make MCMAS the first open source model checker for these specification languages, allowing others to more easily develop further extensions on top of our tool and/or improve its performance. Details of the algorithms used, as well as the implementation, may be found in Chapters 3 and 4 respectively.

[1] For example, the aforementioned `bdd_parameters` struct has two variables for the transition relation `reachRT` and `vRT`; which is actually used when computing preimages depends on whether caching of BDDs is enabled or not. A cleaner solution could involve a separate abstraction for the transition relation, with the actual concrete implementation selected and instantiated at runtime.

2. **First practical algorithm for LDLK model checking.** We modify the construction previously put forth in [41] to devise an alternating automaton construction amenable to a concrete implementation. We prove that our alternative construction is correct, and show how it can be implemented symbolically using the construction of [20]. This algorithm is discussed in Section 5.1.

3. **Formalism of the full branching time extension of Linear Dynamic Logic (LDL), CDL*.** We formalise the natural full branching time extension of LDL, and show that it subsumes both CTL* and LDL. We show that despite its expressive power, CDL* is still PSPACE-complete. Furthermore, we present an algorithm for model checking of CDL* properties, possibly extended with epistemic modalities (CDL*K). This is described in greater detail in Sections 6.1 through 6.3.

4. **First model checking tool for LDLK and CDL*K.** We further extended the functionality of MCMAS to verify properties specified in LDLK and CDL*K. We discuss our symbolic implementation of our LDLK and CDL*K algorithms using binary decision diagrams. We also introduce several performance optimisations and justify their correctness. We are not aware of any existing model checkers that support LDLK or CDL*K. Details of the algorithms used and implementation may be found in Chapters 5 and 6 respectively.

5. **Formalism of the finite trace semantics version of CDL*K (CDL*$_f$K), along with a model checking algorithm.** We formalise the finite trace semantics version of CDL*K, and show that it is PSPACE-complete. We present a reduction of CDL*$_f$K model checking to regular CDL*K model checking and argue its correctness. This is discussed in greater detail in Section 7.2.

6. **First model checking tool for LDLK and CDL*K over finite traces.** We further extended MCMAS to support verification of LDLK and CDL*K properties over finite trace semantics, by implementing the aforementioned reduction of CDL*K model checking over finite traces to that over infinite traces. We then discuss the extension of our tool to verify properties over these semantics. This is discussed in greater detail in Chapter 7.

7. **Comprehensive ISPL test-suite for each specification language.** In order to verify correctness of our implementation, we developed a substantial test-suite of ISPL models and specifications for each of our specification languages. We included a script that automatically invokes MCMAS, parses its output and verifies that the program returns the (manually determined) correct answer. The ISPL files are part of the codebase and may easily be repurposed to check correctness of further extensions to MCMAS, such as performance optimisations and/or attempting alternative verification techniques such as SAT solving or use of sentential decision diagrams. They can also be used to check other model checking tools for multi-agent systems. More detail concerning this testsuite may be found in Section 8.2.

8. **Practical experimental evaluation.** We evaluated how well our implementations of support for the various specification languages scales on a variety of well-known problems in the literature. We also showcased the additional expressivity of the various specification languages, in terms of interesting properties that MCMAS would previously not have been able to verify. This is discussed in greater detail in Sections 8.3 through 8.9.

A more detailed evaluation of our project may be found in Chapter 9.

Chapter 2

Background

In this chapter we introduce various existing techniques from systems verification, as well as several preliminaries from automata theory as well as regular expressions which are relevant to our report. We also include a list of existing model checkers which are related to our work, and discuss in greater depth the existing architecture of the MCMAS model checker [10] which we plan to extend with support for the four additional specification languages.

2.1 Formal Verification Techniques and Model Checking

Formal verification, at a high level, involves the use of mathematical or logical methods to show that a system obeys various functional properties [19]. Typically, this involves:

1. a *framework for modelling systems* such as a description language,
2. a *specification language* allowing properties to be verified to be formally and precisely described, and
3. a *verification method* for determining whether a given system satisfies various properties [54].

We focus on *model checking*, which involves building a model $\mathcal{M}_S$ for a system S under consideration, and expressing each property P as a suitable logical formula ϕ_P. The verification methods we employ involve determining whether a given model $\mathcal{M}_S$ satisfies ϕ_P (written $\mathcal{M}_S \vDash \phi_P$) [54]. Model checking is useful in that it is a (largely) *automatic* technique [30] – while users do need to manually specify some description of S and P, once that is done (generally) no further input from the user is required.

We will examine possible options for each of the three aforementioned components in turn.

2.2 Frameworks for Modelling Systems

2.2.1 Kripke Models

Kripke models are among the most popular frameworks for modelling systems in the context of model-checking temporal properties. We first define several preliminaries and illustrate them with an example:

Definition 2.1. (Kripke Frames) A *Kripke frame* F is a pair (W, R), where W refers to a non-empty set of worlds and R is a binary *accessibility relation* on W (that is, $R \subseteq W \times W$).

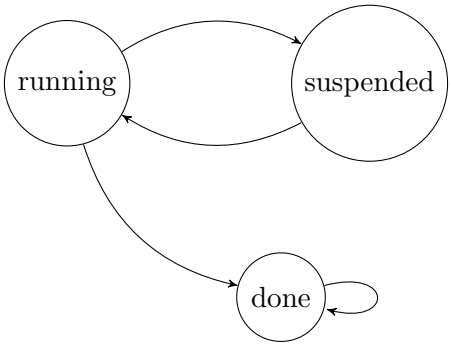


Figure 2.1: Transition system for the Kripke frame F modelling a thread.

Suppose we wish to model a thread in a program. This thread may either be running, suspended or done. If a thread is running, it may complete execution, or be suspended by the operating system scheduler. If a thread has been suspended, it may be scheduled again (and thus become running). Threads that are done remain done. A single thread can be modelled as $F = (W, R)$, where $W = \{\text{running}, \text{suspended}, \text{done}\}$ and $R = \{(\text{running}, \text{suspended}), (\text{suspended}, \text{running}), (\text{running}, \text{done}), (\text{done}, \text{done})\}$. Kripke frames can intuitively be represented as as a transition system or graph, as outlined in Figure 2.1.

Kripke frames do express how a system can evolve, but they do not allow us to (succinctly) discuss properties of the system. This may be done by fixing a set of *atomic propositions* AP, representing properties of interest, and describing the states in which said atomic propositions hold. This results in a *Kripke model*:

Definition 2.2. (Kripke Models) A *Kripke model* is a pair (F, h), where $F = (W, R)$ is a Kripke frame and h is an assignment from atomic propositions AP into the worlds of F (that is, $h : AP \to 2^W$).

Continuing from the previous example, we can define $AP = \{cpu\}$ modelling whether a thread is currently using CPU cycles or not. The truth assignment h then decides which atomic propositions are true at which worlds in W. In this case, a suitable assignment could be $h(cpu) = \{\text{running}\}$ since programs that are suspended or have finished running should not be using the CPU. (F, h) is then a Kripke model for the thread.

Notably, Kripke models only admit a single accessibility relation R. This means that they are not well-suited to handling properties across multiple modalities (such as both temporal and epistemic modalities, which are important for multi-agent systems). A possible solution to this involves allowing multiple accessibility relations to be specified within a single frame or model; this can be done as follows.

Definition 2.3. (Kripke Frames with Multiple Relations) A *Kripke frame with multiple relations* F is a tuple $(W, R_1 \dots R_n)$ for $n \geq 1$, where each R_i for $i \in [1..n]$ is a binary relation on W (that is, $R_i \subseteq W \times W$).

Definition 2.4. (Kripke Models with Multiple Relations) A *Kripke model with multiple relations* M is a tuple (F, h) where F is a Kripke frame with multiple relations and h is an assignment from atomic propositions AP into the worlds of F (that is, $h : AP \to 2^W$).

We can define *equivalence frames* as a special case of Kripke frames with multiple relations, which can be used to evaluate epistemic modalities (that is, modalities about knowledge).

Definition 2.5. (Equivalence Frames) Suppose our system under consideration has n agents. An *equivalence frame* $F = (W, \sim_1 \ldots \sim_n)$ is a tuple where W is a non-empty set of worlds, and for every i in $[1..n]$ $\sim_n$ is an equivalence relation over $W \times W$.

Intuitively, each equivalence relation partitions the set of worlds W into equivalence classes, which represent the possible views of the environment as viewed by the respective agent [62]. An alternative to Kripke models which appears more closely linked to multi-agent systems, as well as captures individual agents' actions is the formalism of *interpreted systems*.

2.2.2 Interpreted Systems

Interpreted systems are a formalism used to model multi-agent systems and reason about knowledge in such systems [75]. Consider the following definition from [64]:

Definition 2.6. (Interpreted Systems) Suppose we have a set of *agents* $\Sigma = \{1 \ldots n\}$ and a special agent known as the *environment*, E. Let AP be a set of *atomic propositions*. Formally, we can describe an *interpreted system* by a tuple $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \right\rangle$, as follows:

- For each agent i, L_i refers to a finite set of *private local states*, and Act_i refers to a finite set of *actions* that the agent can take. These actions must be performed following a *protocol* $P_i : L_i \to 2^{Act_i}$ which indicates which actions are allowable given the private local states of the agent.
- The *environment* E similarly has a finite set of local states L_E, actions Act_E and protocol P_E.
- For each agent i, the evolution of the agents' local states is given by $t_i : L_i \times L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_i$ – in other words, agents' evolution depends only on their local state, the state of the environment and the actions taken by all of the agents. In particular, the evolution of local state is not dependent on other agents' local states. (The environment's evolution is given by $t_E : L_E \times Act_1 \times \ldots \times Act_n \times Act_E \to L_E$.) It is assumed that agents evolve simultaneously.
- The *global state* of the system at a given time instant is given by $g \in L_1 \times \ldots \times L_n \times L_E$. Let G be the set of *reachable* global states (so $G \subseteq (L_1 \times \ldots \times L_n \times L_E)$), and $Act = Act_1 \times \ldots \times Act_n \times Act_E$ be the set of joint actions. Then $t : G \times Act \to G$, the composition of t_i for every i (including the environment), is given by $t(g, a) = g'$ iff $\forall i.t_i(l_i(g), l_E(g), a) = l_i(g')$ where $l_i(g)$ denotes the local state of agent i in g, $a \in Act$, and $\forall i.a_i \in P_i(l_i(g))$ (where $a_i \in Act_i$ and a_i refers to the agent i's action).
- The set G of *reachable* global states can be obtained by considering the evolutions of the system from a set of *initial global states* I.
- $h : AP \to 2^G$ is a valuation function for the atomic propositions in AP, identifying which global states said atomic propositions are true in.

As our work focuses heavily on verifying temporal properties over interpreted systems, we illustrate the above with a concrete example, and one well-known in the literature – the *bit transmission protocol* as described in [65]. Suppose we have a *sender*, which wants to communicate the value of a bit to another agent, a *receiver*. However, the channel they are communicating on exhibits failures; it may fail to deliver a message (in either direction). Once the receiver receives the bit, it begins sending acknowledgments (acks) back to the sender; once the sender receives an ack it stops trying to send the bit. This can be modelled using interpreted systems, as follows:

1. To model the faulty channel, we can assign it local states $L_E = \{R, S, RS, none\}$ corresponding to whether it is able to deliver messages from the Receiver, Sender, both, or neither. The behaviour of the channel itself should be independent of the other agents; this can be established by giving it four actions corresponding to the four local states, a protocol allowing it to act in any of these four actions, independent of the other agents' actions, and an evolution function solely dependent on which action is chosen. In other words $A_E = \{a_R, a_S, a_{RS}, a_{none}\}$, $P_E(l_E) = A_E$ for any $l_E \in L_E$ and $t_E(l_E, a_1, a_2, a_E) = \alpha(a_E)$ where $\alpha(a_{l_E}) = l_E$ for every $l_E \in L_E$.

2. We need four local states for the sender agent (call the sender agent 1), as we need to express whether the bit it is trying to communicate is a 0 or 1, and whether it has received an ack. It has three actions: either it sends a 0, sends a 1 or sends nothing. This can be written $S_1 = \{0, 1, 0_A, 1_A\}$ and $Act_1 = \{send0, send1, none\}$. If the sender has received an ack, its only action is *none*; otherwise, it sends the correct value of the bit – so, $P_1(0) = \{send0\}$, $P_1(1) = \{send1\}$ and $P_1(0_A) = P_1(1_A) = \{none\}$. We return to the evolution function later.

3. We need three local states for the receiver agent (call the receiver agent 2) – either it has not received the bit, or has received the bit (which is either 0 or 1). It has two actions: it either sends nothing, or sends an ack. So this can be written $S_2 = \{none, 0, 1\}$ and $Act_2 = \{none, ack\}$. The receiver sends acks if and only if it has received the bit, so $P_2(none) = \{none\}$ and $P_2(0) = P_2(1) = \{ack\}$.

4. For the evolution function of the sender agent, if the channel is able to deliver messages from the receiver and it has sent an ack, then the sender can receive the ack (transitioning in local state from 0 to 0_A or from 1 to 1_A). Otherwise, it remains in the same state. So,

$$t_1(l_1, l_E, a_1, a_2, a_E) = \begin{cases} 0_A & l_1 = 0 \wedge l_E \in \{R, RS\} \wedge a_2 = ack \\ 1_A & l_1 = 1 \wedge l_E \in \{R, RS\} \wedge a_2 = ack \\ l_1 & \text{otherwise} \end{cases}$$

5. Similarly, for the receiver agent, if the channel is able to deliver messages from the sender and it is sending a bit, then the receiver can receive the bit and update its local state; otherwise, it remains in the same state.

$$t_2(l_2, l_E, a_1, a_2, a_E) = \begin{cases} 0 & l_E \in \{S, RS\} \wedge a_1 = send0 \\ 1 & l_E \in \{S, RS\} \wedge a_1 = send1 \\ l_2 & \text{otherwise} \end{cases}$$

6. The initial global states for this system are the ones in which the sender has not received an ack, and the receiver has not received a bit. We thus have

$$I = \{(l_1, l_2, l_E) : l_1 \in \{0, 1\} \wedge l_2 = none\}$$

7. We can define suitable atomic propositions – for example, if we want to express the idea that the sender has received an ack with the atomic proposition *recack*, this would apply on the reachable global states where $l_1 \in \{0_A, 1_A\}$. In other words,

$$h(recack) = \{(l_1, l_2, l_E) : l_1 \in \{0_A, 1_A\} \wedge (l_1, l_2, l_E) \in G\}$$

This concludes our discussion of interpreted systems. For a more comprehensive treatment, one can consult [64].

2.3 Specification Languages

Many types of modal logics have been proposed and used for model checking various properties over systems – in particular, *temporal* logics concerning time have allowed desirable properties to be expressed, such as *safety* and *liveness* properties [54]. Intuitively, a *safety* property expresses that an undesirable state is never reached; a *liveness* property expresses that some desirable state is reached (possibly eventually, or infinitely often) [66]. Since our work focuses on model checking within the context of multi-agent systems, *epistemic* logics concerning agents' knowledge are also relevant.

In this section we will discuss the syntax, semantics, expressivity and complexity of several popular specification languages. Where appropriate, we will also discuss known algorithms for model checking said specification languages. It is worth mentioning that there tends to be a trade-off between expressivity and model checking complexity. To illustrate the differences in expressivity of the various specification languages, we attempt to translate ten English statements into each logic, as follows.

1. The train will leave the station in the next state.
2. The train will not leave until the doors are closed.
3. The train can leave at some point in the future.
4. It is always possible for the train to leave the station.
5. The train will eventually come to a permanent stop.
6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop.
7. The train always has the blue flag raised in the even states.
8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied.
9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*.
10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop.

2.3.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is a popular specification logic. It treats time as a linear sequence of determined, discrete events [66].

Definition 2.7. (LTL Syntax) The syntax of an LTL formula ϕ_{LTL} is as follows:

$$
\begin{aligned}
\phi_{\text{LTL}} ::= \; & p \text{ where } p \text{ is an atomic proposition} \\
| \; & \top \\
| \; & \neg\phi_{\text{LTL}} \\
| \; & \phi_{\text{LTL}} \wedge \phi_{\text{LTL}} \\
| \; & X\phi_{\text{LTL}} \\
| \; & \phi_{\text{LTL}} U \phi_{\text{LTL}}
\end{aligned}
$$

In addition to the above, we typically take as abbreviations $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$, $F\phi = \top U \phi$ and $G\phi = \neg F \neg \phi$ (where ϕ and ψ are LTL formulae).

In order to define the semantics of LTL formulae in interpreted systems, we need to first define what is meant by a *path* in an interpreted system – it intuitively is a sequence of global states that respect the transition relation of the interpreted system.

Definition 2.8. (Paths and Path Suffixes) Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \right\rangle$ be an interpreted system. A *path* π in $\mathcal{I}$ is an infinite sequence of global states $g_0, g_1, g_2, \ldots$ such that, for all i, $g_i \in G$ and there exists some $a \in Act$ with $g_{i+1} = t(g_i, a)$. Also, we define the i^{th} *state of a path* π by $\pi(i) = g_i$, and the i^{th} *suffix of a path* π by $\pi^i = g_i, g_{i+1}, \ldots$. We say that a path π *starts at* or *starts from* some global state g iff $\pi(0) = g$.

We also introduce shorthand for cyclic paths; suppose we have a path

$$\pi = g_0, g_1, g_2, \ldots, g_k, g_{k+1}, g_{k+2}, \ldots, g_{k+n}, \ldots$$

which after g_{k+n} repeatedly revisits states g_{k+1} through g_{k+n} in the same order (so there exists some nonnegative k and positive n such that $g_{k+n+i} = g_{k+i}$ for all $i \geq 1$). We can write such a path in an abbreviated form $\pi = g_0, g_1, g_2, \ldots, g_k, (g_{k+1}, \ldots, g_{k+n})^\omega$.

We now define the semantics of LTL over interpreted systems, as follows.

Definition 2.9. (LTL Semantics) Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \right\rangle$ be an interpreted system and π a path in $\mathcal{I}$. $\mathcal{I}, \pi \vDash \phi$ may be defined inductively as follows:

- $\mathcal{I}, \pi \vDash p$ iff $\pi(0) \in h(p)$ (for atomic proposition p).
- $\mathcal{I}, \pi \vDash \top$.
- $\mathcal{I}, \pi \vDash \neg\phi$ iff it is not the case that $\mathcal{I}, \pi \vDash \phi$ (written $\mathcal{I}, \pi \nvDash \phi$).
- $\mathcal{I}, \pi \vDash \phi \wedge \psi$ iff $\mathcal{I}, \pi \vDash \phi$ and $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, \pi \vDash X\phi$ iff $\mathcal{I}, \pi^1 \vDash \phi$.
- $\mathcal{I}, \pi \vDash \phi U \psi$ iff $\exists j \geq 0$ such that $\mathcal{I}, \pi^j \vDash \psi$ and for $0 \leq k < j$, $\mathcal{I}, \pi^k \vDash \phi$.

The semantics for the various abbreviations can be derived from the above, though it may be useful to consider them explicitly as well:

- $\mathcal{I}, \pi \nvDash \bot$.
- $\mathcal{I}, \pi \vDash \phi \vee \psi$ iff $\mathcal{I}, \pi \vDash \phi$ or $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, \pi \vDash \phi \rightarrow \psi$ iff $\mathcal{I}, \pi \nvDash \phi$ or $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, \pi \vDash \phi \leftrightarrow \psi$ iff $\mathcal{I}, \pi \vDash \phi$ and $\mathcal{I}, \pi \vDash \psi$, or $\mathcal{I}, \pi \nvDash \phi$ and $\mathcal{I}, \pi \nvDash \psi$.
- $\mathcal{I}, \pi \vDash F\phi$ iff $\exists j \geq 0 . \mathcal{I}, \pi^j \vDash \phi$.
- $\mathcal{I}, \pi \vDash G\phi$ iff $\forall j \geq 0 . \mathcal{I}, \pi^j \vDash \phi$.

Further, for a given global state $g \in G$, $I, g \vDash \phi$ iff ϕ holds on all possible paths starting from g.

Intuitively, $X\phi$ means that ϕ holds in the neXt state and $\phi U\psi$ means that ϕ always holds Until ψ does (and, importantly, at some point ψ becomes true; this is relaxed by the "weak-until" abbreviation sometimes used – $\phi W\psi = (\phi U\psi) \vee G\phi$). Further to this, the common abbreviations $F\phi$ and $G\phi$ mean that ϕ holds at some point in the Future, and that it is Going to be true forever in the future respectively [66].

In terms of expressivity, we revisit the ten statements originally presented and attempt to express them using LTL, where possible.

1. The train will leave in the next state: X(train leaves).
2. The train will not leave until the doors are closed: (¬(train leaves)U(close doors)). (Note: This example could arguably be written with weak until instead, though it is probably desirable that the train is eventually able to leave.)
3. The train can leave at some point in the future: Not possible, as LTL always quantifies over *all paths* and this statement is attempting to express that *some* path on which the train permanently leaves the station exists. It is worth mentioning that we can express the negation of this property as G¬(train leaves); if this is satisfied on all paths then the original statement is false, so we can partially alleviate this limitation [54].
4. It is possible for the train to permanently leave the station: This property mixes existential and universal path quantifiers and thus cannot be expressed in LTL; the complement will still contain both existential and universal path quantifiers (albeit reversed) and so we cannot use the technique for statement 3.
5. The train will eventually come to a permanent stop: FG(stop).
6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop: Not possible; consider statement 4.
7. The train always has the blue flag raised in the even states: This cannot be expressed in LTL – we can approximate it with (blue) $\wedge$ XX(blue) etc., but to truly capture it with this approach would require a formula of infinite size.
8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: Not possible as LTL does not have epistemic modalities.
9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*: As in statement 8, not possible as LTL does not have epistemic modalities.
10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop: Not possible, for varying reasons – consider statements 4, 7 and 8.

In particular, LTL is unable to express properties concerning the existence of paths originating from a state (it assumes that we consider all paths). It is, of course, also unable to express properties concerning modalities other than time.

There are several algorithms for model checking LTL. Typically, these entail either

- the construction of an explicit *Büchi automaton* corresponding to the model and LTL specification, and checking it for nonemptiness [45], or

- a *tableau construction* algorithm which generates a symbolic Büchi automaton; composing this automaton with the original model yields a composition under which the original LTL specification holds if and only if a different CTL specification with fairness constraints also holds [28].

Existing studies have generally shown that the latter approach tends to be superior, as it can be implemented symbolically [78]. As LTL is clearly a key point of focus of our project, we discuss these algorithms in considerably greater detail in Section 2.5. In general, model checking for LTL is PSPACE-complete; algorithms generally require time linear in the size of the model but exponential in the size of the specification being checked. More formally, we introduce *big-O notation*, following the definition from [33].

Definition 2.10. (Big-O Notation) We say that $f(n) = O(g(n))$ if there exists some positive c and n_0, such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. For example, $f(n) = n^2 + n$ is $O(n^2)$, because we can choose $n_0 = 1$ and $c = 2$; consider that for every $n_0 > 1$, $n^2 > n$ and so $f(n) = n^2 + n \leq n^2 + n^2 = 2n^2$. But $f(n)$ is not $O(n)$ – this can be shown via a proof by contradiction. Supposing it was, there must exist some c and n_0 such that $0 \leq n^2 + n \leq cn$ for every $n > n_0$. However, $n^2 + n \leq cn$ holds iff $n + 1 \leq c$ holds (since n is positive), and thus the inequality only holds for $n \leq c - 1$ i.e. it cannot be the case that it holds for arbitrarily large n.

Thus, we can write the aforementioned complexity as $O\left(|\mathcal{I}|2^{|\phi|}\right)$ [28].

2.3.2 Computation Tree Logic (CTL)

Computation Tree Logic (CTL) is another popular specification logic. Unlike LTL, it is a *branching-time* logic, modelling time as a tree structure with multiple possible paths [54]. An advantage of CTL is that it allows explicit quantification over paths [66].

Definition 2.11. (CTL Syntax) The syntax of a CTL formula ϕ_{CTL} is as follows:

$\phi_{\text{CTL}} ::= p$ where p is an atomic proposition
$\mid \top$
$\mid \neg\phi_{\text{CTL}}$
$\mid \phi_{\text{CTL}} \wedge \phi_{\text{CTL}}$
$\mid EX\phi_{\text{CTL}}$
$\mid EG\phi_{\text{CTL}}$
$\mid E(\phi_{\text{CTL}} U \phi_{\text{CTL}})$

In addition to the above, suppose ϕ and ψ are CTL formulae. We typically take as abbreviations from propositional logic $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. The remainder of the temporal connectives for CTL can be defined from the aforementioned three, as follows. First observe from [54] that

$$A(\phi U \psi) = \neg(E((\neg\psi)U(\neg\phi \wedge \neg\psi)) \vee EG\neg\psi)$$

We can then define $AF\phi = A(\top U \phi)$, $EF\phi = E(\top U \phi)$ and $AG\phi = \neg EF\neg\phi$. Finally $AX\phi = \neg EX\neg\phi$.

We can now define the semantics of CTL over interpreted systems. Notice that these semantics are defined directly over states.

Definition 2.12. (CTL Semantics) Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \right\rangle$ be an interpreted system and g a global state in $\mathcal{I}$. $\mathcal{I}, g \vDash \phi$ may be defined inductively, as follows:

- $\mathcal{I}, g \vDash p$ iff $g \in h(p)$ for atomic proposition p.
- $\mathcal{I}, g \vDash \top$.
- $\mathcal{I}, g \vDash \neg\phi$ iff it is not the case that $\mathcal{I}, g \vDash \phi$.
- $\mathcal{I}, g \vDash \phi \wedge \psi$ iff $\mathcal{I}, g \vDash \phi$ and $\mathcal{I}, g \vDash \psi$.
- $\mathcal{I}, g \vDash EX\phi$ iff there exists some path π in $\mathcal{I}$ starting at g, such that $\mathcal{I}, \pi(1) \vDash \phi$.
- $\mathcal{I}, g \vDash EG\phi$ iff there exists some path π in $\mathcal{I}$ starting at g, such that $\forall i \geq 0$ we have $\mathcal{I}, \pi(i) \vDash \phi$.
- $\mathcal{I}, g \vDash E(\phi U \psi)$ iff there exists some path π in $\mathcal{I}$ starting at g, such that $\exists i \geq 0$ such that $\mathcal{I}, \pi(i) \vDash \psi$ and for all $0 \leq j < i$ we have $\mathcal{I}, \pi(j) \vDash \phi$.

As before, it may be useful to explicitly set out the semantics for the abbreviations.

- $\mathcal{I}, g \nvDash \bot$.
- $\mathcal{I}, g \vDash \phi \vee \psi$ iff $\mathcal{I}, g \vDash \phi$ or $\mathcal{I}, g \vDash \psi$.
- $\mathcal{I}, g \vDash \phi \rightarrow \psi$ iff $\mathcal{I}, g \nvDash \phi$ or $\mathcal{I}, g \vDash \psi$.
- $\mathcal{I}, g \vDash \phi \leftrightarrow \psi$ iff $\mathcal{I}, g \vDash \phi$ and $\mathcal{I}, g \vDash \psi$, or $\mathcal{I}, g \nvDash \phi$ and $\mathcal{I}, g \nvDash \psi$.
- $\mathcal{I}, g \vDash EF\phi$ iff there exists some path π in $\mathcal{I}$ starting at g, such that $\exists i \geq 0$ with $\mathcal{I}, \pi(i) \vDash \phi$.
- $\mathcal{I}, g \vDash AX\phi$ iff on every path π in $\mathcal{I}$ starting at g we have $\mathcal{I}, \pi(1) \vDash \phi$.
- $\mathcal{I}, g \vDash AG\phi$ iff on every path π in $\mathcal{I}$ starting at g, $\forall i \geq 0$ we have $\mathcal{I}, \pi(i) \vDash \phi$.
- $\mathcal{I}, g \vDash A(\phi U \psi)$ iff on every path π in $\mathcal{I}$ starting at g, $\exists i \geq 0$ such that $\mathcal{I}, \pi(i) \vDash \psi$ and for all $0 \leq j < i$ we have $\mathcal{I}, \pi(j) \vDash \phi$.
- $\mathcal{I}, g \vDash AF\phi$ iff on every path π in $\mathcal{I}$ starting at g, $\exists i \geq 0$ with $\mathcal{I}, \pi(i) \vDash \phi$.

Intuitively, the temporal connectives' first letters indicates whether they are concerned with evolution 'along All paths' or 'along at least (there Exists) one path' [54]. The second letters, as in LTL, suggest the property holding in the neXt state, Going to hold forever, one property holding Until another becomes true, or holding at some point in the Future [66].

We turn our attention to the expressivity of CTL – we revisit the ten statements originally presented and now try to express them in CTL.

1. The train will leave in the next state: AX(train leaves).
2. The train will not leave until the doors are closed: $A(\neg$(train leaves)U(close doors)).
3. The train can leave at some point in the future: EF(train leaves) – 'there exists a path on which, at some point in the future, the train leaves'.
4. It is possible for the train to permanently leave the station: $EF(AG$(train left)).

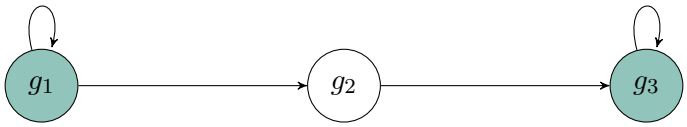


Figure 2.2: Counterexample state space for $AF(AG(\text{stop}))$ and statement 5; the nodes at which stop holds are shaded green.

5. The train will eventually come to a permanent stop: Not possible [79]. It seems that $AF(AG(\text{stop}))$ might work, but it actually does not – consider a simple interpreted system $\mathcal{I}$ with three global states and a suitable global evolution as represented in Figure 2.2, where $I = \{g_1\}$ and $h(\text{stop}) = \{g_1, g_3\}$. Notice that $\mathcal{I}, g_1 \not\models AF(AG(\text{stop}))$. Consider the path looping infinitely in g_1; we can write this as $(g_1)^\omega$ using the shorthand from Definition 2.8. Clearly $AG(\text{stop})$ does not hold in any state along this path, as g_2 can always be a successor of g_1, giving us $\mathcal{I}, g_1 \not\models AF(AG(\text{stop}))$. However, the train *will* eventually come to a permanent stop if the system is in state g_1; either it stays permanently in g_1, or it transitions to g_2, after which it must transition to g_3 and loop forever in g_3.

6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop: Not possible; consider statement 5.

7. The train always has the blue flag raised in the even states: This cannot be expressed in CTL. As before, we can approximate it with $(\text{blue}) \wedge AXAX(\text{blue})$ etc., but to capture it with this approach would again require a formula of infinite size.

8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: Not possible as CTL does not have epistemic modalities.

9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*: As in statement 8, not possible as CTL does not have epistemic modalities.

10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop: Not possible, for varying reasons – consider statements 7 and 8.

While CTL is able to quantify explicitly over paths, it is restricted in that each temporal operator is necessarily preceded by a path quantifier. Consequently, there can be difficulties with describing behaviours that may arise on different branches at different times [79] (such as in our statement 5). How significant this limitation is in an industrial context is certainly contentious [95]. Like LTL, CTL is also a temporal logic and thus on its own cannot express properties concerning non-temporal modalities.

Several methods exist for model checking a CTL formula ϕ. Typically, we will often want to determine if a property is satisfied from the initial states of our system $\mathcal{I}$; in other words, whether $I \subseteq [\phi]_\mathcal{I}$, where $[\phi]_\mathcal{I}$ is the set of global states in which ϕ holds (that is, $[\phi]_\mathcal{I} = \{g \in G : \mathcal{I}, g \vDash \phi\}$). However, it is often algorithmically more convenient to simply compute $[\phi]_\mathcal{I}$ and then check whether $I \subseteq [\phi]_\mathcal{I}$ [66]. Typically, this involves recursively computing the states at which the various subformulas of ϕ hold, and synthesising these results appropriately when computing ϕ. Alternatives do exist, such as automata construction [17]. For example, given

$[\phi]_{\mathcal{I}}$, computing $[EX\phi]_{\mathcal{I}}$ involves finding all states that can possibly evolve into any of the states in $[\phi]_{\mathcal{I}}$.

Definition 2.13. (Existential and Universal Preimages)
Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle$ be an interpreted system and $X \subseteq G$. The *existential preimage* of X refers to the set of global states in G which can possibly evolve into a state in X; that is, $\text{pre}_\exists(X) = \{g \in G : \exists a \in Act.t(g,a) = g' \wedge g' \in X\}$. The *universal preimage* of S is defined similarly as the set of global states in G which necessarily evolve into a state in X; $\text{pre}_\forall(X) = \{g \in G : \forall g' \in t(g,a).g' \in X\}$. Notice that $\text{pre}_\forall(X) = G - \text{pre}_\exists(G - X)$.

Clearly, this gives us the facility to check EX and AX; through the use of fix-point computations, this allows us to check the other temporal modalities as well. We outline two helper procedures that can be used in model checking CTL formulae of the form $AF\phi$ and $E(\phi U \psi)$.

Definition 2.14. (Fixed Points) Let $f : S \to S$ be a function and S be a set. Let $X \subseteq S$. Then, X is a *fixed point* of f iff $f(X) = X$. X is the *least fixed point* of f iff X is a fixed point and, for every fixed point V of f, $X \subseteq V$. The least fixed point of f may be written as $\text{LFP}(f)$. Conversely, X is the *greatest fixed point* of f iff X is a fixed point, and for every fixed point V of f, $V \subseteq X$.

Definition 2.15. (SAT_{AF} and SAT_{EU}) We define $SAT_{AF}(\phi) = \text{LFP}(f)$ where $f(X) = [\phi]_{\mathcal{I}} \cup \text{pre}_\forall(X)$. Intuitively, $AF\phi$ is true in the states where ϕ already holds, or in the states from which all evolutions lead to states in which $AF\phi$.

We also define $SAT_{EU}(\phi, \psi) = \text{LFP}(g)$ where $g(X) = [\psi]_{\mathcal{I}} \cup ([\phi]_{\mathcal{I}} \cap \text{pre}_\exists(X))$. Intuitively, $E(\phi U \psi)$ holds in the states where ψ is true, or in the states where ϕ is true *and* there exists an evolution to a state where $E(\phi U \psi)$ holds. Please consult [66] for a more detailed explanation as to why these are correct.

We can now compute $[\phi]_{\mathcal{I}}$ inductively, depending on the principal connective of ϕ. A possible method is as follows (adapting the algorithm from [66] for interpreted systems):

1. $[\top]_{\mathcal{I}} = G$.
2. $[\bot]_{\mathcal{I}} = \emptyset$.
3. $[p]_{\mathcal{I}} = h(p)$ (where p is an atomic proposition).
4. $[\neg\phi]_{\mathcal{I}} = G - [\phi]_{\mathcal{I}}$.
5. $[\phi \wedge \psi]_{\mathcal{I}} = [\phi]_{\mathcal{I}} \cap [\psi]_{\mathcal{I}}$.
6. $[\phi \vee \psi]_{\mathcal{I}} = [\phi]_{\mathcal{I}} \cup [\psi]_{\mathcal{I}}$.
7. $[\phi \to \psi]_{\mathcal{I}} = [\neg\phi]_{\mathcal{I}} \cup [\psi]_{\mathcal{I}}$.
8. $[\phi \leftrightarrow \psi]_{\mathcal{I}} = ([\phi]_{\mathcal{I}} \cap [\psi]_{\mathcal{I}}) \cup ([\neg\phi]_{\mathcal{I}} \cap [\neg\psi]_{\mathcal{I}})$.
9. $[EX\phi]_{\mathcal{I}} = \text{pre}_\exists([\phi]_{\mathcal{I}})$.
10. $[EG\phi]_{\mathcal{I}} = [\neg AF\neg\phi]_{\mathcal{I}}$.
11. $[E(\phi U \psi)]_{\mathcal{I}} = SAT_{EU}(\phi, \psi)$.
12. $[EF\phi]_{\mathcal{I}} = [E(\top U \phi)]_{\mathcal{I}}$.

13. $[AX\phi]_{\mathcal{I}} = \text{pre}_\forall([\phi]_{\mathcal{I}})$.
14. $[AG\phi]_{\mathcal{I}} = [\neg EF\neg\phi]_{\mathcal{I}}$.
15. $[A(\phi U\psi)]_{\mathcal{I}} = [\neg(E((\neg\psi)U(\neg\phi \wedge \neg\psi) \vee EG\neg\psi)]_{\mathcal{I}}$.
16. $[AF\phi]_{\mathcal{I}} = SAT_{AF}(\phi)$.

In general, CTL model checking can be carried out in linear time with respect to both the size of the interpreted system $\mathcal{I}$ under consideration and the specification ϕ being considered. In other words, model checking can be done in $O(|\mathcal{I}||\phi|)$ time [84].

2.3.3 Full Branching Time Logic (CTL*)

Full Branching Time Logic (CTL*) is a more general temporal logic, in which we allow path quantifiers to prefix *unrestricted combinations* of the LTL temporal modalities [40]. We present the syntax of CTL* below.

Definition 2.16. (Path Formula) The syntax of a *path formula* ψ is as follows:

$\psi ::=$ ϕ where ϕ is a state formula; this will be defined shortly.
| $\neg\psi$
| $\psi \wedge \psi$
| $X\psi$
| $\psi U\psi$

We also include the abbreviations from propositional logic and for the temporal modalities from LTL (that is, where ϕ and ψ are path formulas, $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$, $F\phi = \top U\phi$ and $G\phi = \neg F\neg\phi$).

Definition 2.17. (State Formulas) The syntax of a *state formula* ϕ is as follows:

$\phi ::=$ p where p is an atomic proposition
| $\top$
| $\neg\phi$
| $\phi \wedge \phi$
| $E\psi$ where ψ is a path formula

As before, we also include the abbreviations from propositional logic. (That is, where ϕ and ψ are state formulas, $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$ and $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$). We also add an additional abbreviation, $A\phi = \neg E\neg\phi$.

Definition 2.18. (CTL* formulas) When not otherwise qualified, a CTL* *formula* refers to a state formula, as defined in Definition 2.17.

The semantics of CTL* may be defined as follows:

Definition 2.19. (CTL* semantics) Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle$ be an interpreted system. Let $g \in G$ be a global state of the system and π be a path in $\mathcal{I}$. Then,

- $\mathcal{I}, g \vDash E\psi$ iff there exists a path π starting at g such that $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, g \vDash A\psi$ iff on all paths π starting at g, we have $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, g \vDash \phi$ otherwise follows the semantics of CTL.

- $\mathcal{I}, \pi \vDash \phi$ iff $\mathcal{I}, \pi(0) \vDash \phi$ (where ϕ is a state formula).
- $\mathcal{I}, \pi \vDash \psi$ otherwise follows the semantics of Ltl.

Clearly, Ctl* subsumes both Ltl and Ctl [40]; for any given Ltl or Ctl formula ψ or ϕ respectively, it is possible to rewrite said formula as an equivalent Ctl* formula. Consider the Ltl formula ψ; we say that $\mathcal{I}, g \vDash \psi$ in Ltl iff on every path starting at g we have $\mathcal{I}, \pi \vDash \psi$. This precisely matches the semantics for the Ctl* formula $A\psi$.

On the other hand, consider the Ctl formula ϕ; we can clearly generate an equivalent formula in Ctl* by immediately invoking the appropriate modality when building path formulae. More formally, Ctl can be seen as the subset of Ctl* where path formulae are restricted to $X\phi, \phi U \phi, F\phi$ and $G\phi$ where ϕ is a state formula.

However, Ctl* is clearly strictly more expressive than both Ctl and Ltl. Most obviously, it can express combinations of properties that can only be represented in one or the other. We return to the ten statements initially presented, and (attempt to) express them in Ctl*.

1. The train will leave in the next state: $A(X(\text{train leaves}))$ (brackets added for emphasis of the difference between Ctl and Ctl*).
2. The train will not leave until the doors are closed: $A(\neg(\text{train leaves})U(\text{close doors}))$.
3. The train can leave at some point in the future: $E(F(\text{train leaves}))$ – 'there exists a path on which, at some point in the future, the train leaves'.
4. It is possible for the train to permanently leave the station: $E(F(A(G(\text{train left}))))$.
5. The train will eventually come to a permanent stop: $A(FG\text{stop})$.
6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop: $E(F(A(G(\text{train left})))) \wedge A(FG\text{stop})$. This is simply the conjunction of statements 4 and 5, though we were previously not able to express this since neither Ltl nor Ctl could express both.
7. The train always has the blue flag raised in the even states: This still cannot be expressed in Ctl*. As before, it seems we can approximate it with $A((\text{blue}) \wedge XX(\text{blue})\ldots)$ but a formula of infinite size would still be required with this approach.
8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: Not possible as Ctl* does not have epistemic modalities.
9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*: As in statement 8, not possible as Ctl* does not have epistemic modalities.
10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop: Not possible, for varying reasons – consider statements 7 and 8.

Thus Ctl* seems quite expressive; it is able to represent a fairly wide variety of temporal properties. Nonetheless, there are still several potentially interesting temporal properties (statement 7) that are not expressible in Ctl*.

Model checking CTL* can be done by a reduction to the LTL model checking techniques discussed in Section 2.3.1 and elaborated upon in Section 2.5. This is typically done with a 'recursive descent' approach; see [39] for more details. The complexity is similar to that for LTL; exponential in the formula size and linear in the model size (in terms of computational complexity, model checking CTL* is also PSPACE-complete) [39]. Nonetheless, despite subsuming LTL and having the same worst-case computational complexity, in practice verifying properties in CTL* is often slower than in LTL [84].

2.3.4 Linear Dynamic Logic (LDL)

Linear Dynamic Logic (LDL) is a relatively new logic, proposed by Vardi in 2011 as a replacement for LTL [96]. The syntax of LDL formulas is as follows.

Definition 2.20. (LDL Syntax) An LDL *formula* ϕ is constructed by the following grammar:

$\phi ::= p$ where p is an atomic proposition
$\mid \top$
$\mid \neg\phi$
$\mid \phi \wedge \phi$
$\mid \langle\rho\rangle\, \phi$

$\rho ::= \psi$ where ψ is a propositional formula
$\mid \phi?$
$\mid \rho + \rho$
$\mid \rho; \rho$
$\mid \rho^*$

We include the usual propositional abbreviations for LDL formulas ϕ; that is, $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$. We also add a modal abbreviation $[\rho]\phi = \neg\langle\rho\rangle\neg\phi$. As may be apparent, the syntax of LDL borrows heavily from that of *Propositional Dynamic Logic* (PDL), a popular specification language for programs [35].

We now define the semantics for LDL over interpreted systems.

Definition 2.21. (LDL semantics) Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \right\rangle$ be an interpreted system. Let π be a path in $\mathcal{I}$. Then,

- $\mathcal{I}, \pi \vDash p$ iff $\pi(0) \in h(p)$.
- $\mathcal{I}, \pi \vDash \top$.
- $\mathcal{I}, \pi \vDash \neg\phi$ iff it is not the case that $\mathcal{I}, \pi \vDash \phi$.
- $\mathcal{I}, \pi \vDash \phi \wedge \psi$ iff $\mathcal{I}, \pi \vDash \phi$ and $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, \pi \vDash \langle\rho\rangle\, \phi$ iff there exists some $i \geq 0$ such that $(0, i) \in \mathcal{R}(\rho, \pi)$ and $\mathcal{I}, \pi^i \vDash \psi$.

The relation $\mathcal{R}$ is inductively defined, as follows:

- $\mathcal{R}(\psi, s) = \{(i, i+1) : \pi^i \vDash \psi\}$ (where ψ is a propositional formula)
- $\mathcal{R}(\phi?, s) = \{(i, i) : \pi^i \vDash \phi\}$
- $\mathcal{R}(\rho + \rho', s) = \mathcal{R}(\rho, s) \cup \mathcal{R}(\rho', s)$

- $\mathcal{R}(\rho; \rho', s) = \{(i,j) : \exists k \text{ s.t. } (i,k) \in \mathcal{R}(\rho, s) \wedge (k,j) \in \mathcal{R}(\rho', s)\}$
- $\mathcal{R}(\rho^*, s) = \{(i,i)\} \cup \{(i,j) : \exists k \text{ s.t. } (i,k) \in \mathcal{R}(\rho, s) \wedge (k,j) \in \mathcal{R}(\rho^*, s)\}$

Analogous to LTL, for a global state $g \in G$, we have $\mathcal{I}, g \vDash \phi$ iff on every path π starting at g we have $\mathcal{I}, \pi \vDash \phi$.

Intuitively, the path expressions ρ in the dynamic modalities behave similarly to regular expressions, in that they take on the following meanings:

- ψ checks if the propositional formula ψ holds at the current instant along this path.
- $\phi?$ checks if the LDL formula ϕ holds at the current instant along this path. Unlike ψ, though, we do not consider the evaluation of $\phi?$ to "progress" along the path, as we would with ψ.
- $\rho + \rho'$ models *nondeterministic selection* between two path expressions; it checks whether either of the path expressions ρ or ρ' is satisfied by this path.
- $\rho; \rho'$ models *sequential composition* of two path expressions; it checks whether it is possible for the current path to satisfy ρ *and then* ρ'.
- $\langle \rho^* \rangle$ models *zero or more instances of* ρ, much like the Kleene star for regular expressions.

The semantics of $\mathcal{I}, \pi \vDash \langle \rho \rangle \phi$ then intuitively mean that some prefix of π matches the path expression ρ, and then the remaining suffix of π satisfies ϕ. For the dual operator we have $\mathcal{I}, \pi \vDash [\rho]\phi$ iff for every prefix of π matching the path expression ρ, the remaining suffix of π satisfies ϕ.

Thus, for example, supposing p, q and r are atomic propositions, the LDL formula $\langle p?; q; (p+q) \rangle r$ holds on a path π if we currently have both p and q holding (otherwise $p?; q$ is not satisfied), followed by a state where *either* p or q holds, followed by a state where r holds. We now demonstrate the expressivity of LDL by attempting to translate the ten English statements to LDL.

1. The train will leave the station in the next state: $\langle \top \rangle$(train leaves). Notice that any single state matches the propositional formula $\top$, so we are looking at whether (train leaves) holds in the *next* state, as desired. $[\top]$(train leaves) also has the same semantics, since the path expression $\top$ can also only possibly match one state.
2. The train will not leave until the doors are closed: $\langle (\neg(\text{train leaves}))^* \rangle$(doors closed). This holds if the doors are closed immediately; otherwise, for the formula to be satisfied, in every state before the doors are closed, the train must not leave. (This assumes the "strong until" reading of the statement; if the train doors are never closed and the train never leaves the specification is false. We can construct a "weak until" version: $\langle \neg(\text{train leaves})^* \rangle(\text{doors closed}) \vee [\top^*] \neg(\text{train leaves})$.)
3. The train can leave at some point in the future: Not possible, since LDL quantifies over all paths. We can express the negation of this property as $\neg \langle \top^* \rangle$(train leaves) and use the technique mentioned in [54] to check whether this property holds.
4. It is always possible for the train to leave the station: Not possible, and we cannot use the technique for statement 3 as well since this involves both existential and universal path quantifiers.

5. The train will eventually come to a permanent stop: $\langle\top^*\rangle[\top^*](\text{stop})$. This holds if there exists some point after which after *any number* of additional states the train will stop, which is captured by the LDL formula.

6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop: Not possible; consider that we could not express statement 4.

7. The train always has the blue flag raised in the even states: $[(\top;\top)^*](\text{blue flag raised})$. Since any even length sequence matches $(\top;\top)^*$ we need (blue flag raised) to hold after any even length sequence i.e. in all of the even states.

8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: Not possible as LDL focuses purely on time; it does not have epistemic modalities.

9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*. Not possible as LDL focuses purely on time; it does not have epistemic modalities.

10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop: Not possible, for varying reasons – consider statements 4 and 8.

We thus observe that the expressivity of LDL and CTL* are incomparable; there are statements in each one that are not expressible in the other (statement 7 was expressible in LDL but not in CTL*, while statements 4 and 6 were expressible in CTL* but not in LDL). It has been shown that the properties expressible in LDL correspond to the properties expressible in monadic second-order logic [96], and hence also to the ω-regular properties [24].

LDL maintains the *exponential-compilation property* of LTL; that is, we can compile an LDL formula into a Büchi automaton with size exponential in the formula length [96]. This implies that the model checking problem for LDL is, like LTL, PSPACE-complete [41]. While we are not aware of any existing model checkers for LDL, an alternating automaton construction for checking LDL exists [41].

2.3.5 Epistemic Logic and Linear Temporal Logic (LTLK)

The previous four logics are *temporal logics* – that is, they focus on properties of a system over time. However, within the context of multi-agent systems we are often concerned with other modalities, in particular the *knowledge* that agents may possess[1]. We typically pair epistemic logic with a temporal specification language, to allow specifications about agents' knowledge over time. In this section we discuss the addition of epistemic modalities to linear temporal logic, giving rise to the temporal-epistemic specification language LTLK. It should be noted that the other temporal specification languages can also be enriched with epistemic modalities, with varying expressivity and complexity.

Definition 2.22. (LTLK Syntax)
Let $\mathcal{I}$ be an interpreted system; $\mathcal{I} = \big\langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E\}}, I, h\big\rangle$. The syntax of an LTLK formula ϕ_{LTLK} is as follows:

[1] Other logics such as deontic logic or alternating-time temporal logic (ATL) focus on other relevant properties, such as rules agents should follow and strategies agents can pursue, respectively.

$\phi_{\text{LTLK}} ::= p$ where p is an atomic proposition
$\mid \top$
$\mid \neg\phi_{\text{LTLK}}$
$\mid \phi_{\text{LTLK}} \wedge \phi_{\text{LTLK}}$
$\mid X\phi_{\text{LTLK}}$
$\mid \phi_{\text{LTLK}} U \phi_{\text{LTLK}}$
$\mid K_i\phi_{\text{LTLK}}$ where $i \in \Sigma$
$\mid E_\Gamma\phi_{\text{LTLK}}$ where $\Gamma \subseteq \Sigma$
$\mid D_\Gamma\phi_{\text{LTLK}}$ where $\Gamma \subseteq \Sigma$
$\mid C_\Gamma\phi_{\text{LTLK}}$ where $\Gamma \subseteq \Sigma$

Let ϕ and ψ be LTLK formulae. We add the standard propositional abbreviations as well; that is, $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$, and the temporal abbreviations from plain LTL: $F\phi = \top U \phi$ and $G\phi = \neg F \neg \phi$.

The semantics of LTLK for the non-epistemic operators are the same as that of LTL. We define the semantics for the epistemic operators as follows.

Definition 2.23. (LTLK Semantics)
Let $\mathcal{I}$ be an interpreted system; $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle$. Let $g \in G$ be a global state of $\mathcal{I}$ and π be a path in $\mathcal{I}$. Define $l_i : G \rightarrow L_i$ as the function returning the local state of the i^{th} agent from the input global state. Then,

- $\mathcal{I}, \pi \vDash K_i\phi$ iff for every path ρ where $l_i(\pi(0)) = l_i(\rho(0))$ we have $\mathcal{I}, \rho \vDash \phi$.
- $\mathcal{I}, \pi \vDash E_\Gamma\phi$ iff for every path ρ where $l_i(\pi(0)) = l_i(\rho(0))$ for *some* agent $i \in \Gamma$ we have $\mathcal{I}, \rho \vDash \phi$.
- $\mathcal{I}, \pi \vDash D_\Gamma\phi$ iff for every path ρ where $l_i(\pi(0)) = l_i(\rho(0))$ for *all* agents $i \in \Gamma$ we have $\mathcal{I}, \rho \vDash \phi$.
- $\mathcal{I}, \pi \vDash C_\Gamma\phi$ iff for every path ρ where $\rho_1, \rho_2, \ldots$ are paths such that $l_i(\pi(0)) = l_{k_1}(\rho_1(0)) = l_{k_2}(\rho_2(0)) = \ldots = l_j(\rho(0))$ for agents $i, j, k_1, \ldots \in \Gamma$, we have $\mathcal{I}, \rho \vDash \phi$.
- The semantics for the temporal and propositional connectives, and for atoms, follows that of LTL.

LTLK satisfaction over states is defined in a similar way to LTL satisfaction over states; we say that $\mathcal{I}, g \vDash \phi$ iff $\mathcal{I}, \pi \vDash \phi$ for *every* path π starting at g.

We consider private knowledge as a function of local states [66]. $K_i\phi$ means "the agent i knows ϕ"; that is, in every possible global state g where $l_i(g)$ matches the agent's local state, ϕ holds. Further to this, the intuitive reading of the other epistemic modalities are as follows [66]:

- $E_\Gamma\phi$ is defined on the union of the equivalence relations for the agents in Γ; based on this we can conclude $E_\Gamma\phi = \bigwedge_{i \in \Gamma} K_i\phi$. Thus $E_\Gamma\phi$ may be read as 'all agents (Everybody) in the group Γ knows ϕ'.
- $D_\Gamma\phi$ is defined on the intersection of the equivalence relations for the agents in Γ; it appears impossible to define this in terms of $K_i\phi$. It can be read as 'it is Distributed knowledge in the group Γ that ϕ'. Intuitively, if $D_\Gamma\phi$ holds and the agents in Γ share their knowledge, they can collectively determine that ϕ holds.

- $C_\Gamma\phi$ is defined on the transitive closure of the equivalence relations for the agents in Γ; this is equivalent to the unbounded conjunction $E_\Gamma\phi \wedge E_\Gamma E_\Gamma\phi \wedge \ldots$. A typical reading would be that 'it is Common knowledge in the group Γ that ϕ'.

The above semantics are known as *observational semantics*; they do not consider that agents may, in practice, be aware of how much time has passed (*clock semantics*) or their previous states (*perfect recall*) [53]. We choose these semantics as they do not affect the complexity of the model-checking problem, as opposed to clock semantics (leading to complexity exponential in the size of the model, thereby significantly inhibiting scalability) or perfect recall (which is undecidable in general) [53].

In terms of expressivity, clearly as far as temporal properties are concerned LTLK does not add to LTL; revisiting the ten statements presented at the beginning, clearly its performance on statements 1 through 7, as well as 10, remains the same. However, it does allow us to consider and evaluate statements 8 and 9, which require specifications to capture epistemic modalities.

8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: $G((\text{A in tunnel}) \rightarrow K_B(\text{tunnel occupied}))$.

9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*: $G(K_A(\text{B in tunnel}) \rightarrow C_\Gamma K_A(\text{B in tunnel}))$ where $\Gamma = \{A, B\}$. We assume here that knowledge is introspective (that is, an agent knows that it knows what it knows; $K_i\phi \rightarrow K_iK_i\phi$ is assumed valid).

Model checking of epistemic modalities on their own can be done by computing the states in which the relevant subformula ϕ holds and then appropriately transforming this set of states depending on the epistemic accessibility relations [66]; for model checking of generic LTLK formulae, please consult Section 3.1. We introduce a preliminary definition that will be useful:

Definition 2.24. (Epistemic Accessibility) Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E\}}, I, h\rangle$ be an interpreted system. Let $g, g' \in G$ be two global states of G. Then, we define the epistemic accessibility relation for agent i, $\sim_i \subseteq G \times G$ by $(g, g') \in G$ iff $l_i(g) = l_i(g')$. We also define accessibility relations corresponding to the other epistemic modalities:

- $\sim_\Gamma^E \subseteq G \times G$ by $(g, g') \in G$ iff $\exists i \in G$ s.t. $l_i(g) = l_i(g')$
- $\sim_\Gamma^D \subseteq G \times G$ by $(g, g') \in G$ iff $\forall i \in G, l_i(g) = l_i(g')$
- $\sim_\Gamma^C \subseteq G \times G = \text{LFP}(f)$ where $f(X) = \sim_\Gamma^E \cup \{(g, g'') : \exists (g, g') \in X \text{ s.t. } (g', g'') \in \sim_\Gamma^E\}$

We briefly outline how, given the set of states $[\phi]_\mathcal{I} \subseteq G$ satisfying a formula ϕ, we can evaluate the epistemic modalities over ϕ. Suppose $S = [\phi]_\mathcal{I}$.

- $[K_i\phi]_\mathcal{I} = G - \{g : \exists g' \in (G - S) \wedge \sim_i (g, g')\}$.
- $[E_\Gamma\phi]_\mathcal{I} = G - \{g : \exists g' \in (G - S) \wedge \sim_\Gamma^E (g, g')\}$.
- $[D_\Gamma\phi]_\mathcal{I} = G - \{g : \exists g' \in (G - S) \wedge \sim_\Gamma^D (g, g')\}$.
- $[C_\Gamma\phi]_\mathcal{I} = G - \{g : \exists g' \in (G - S) \wedge \sim_\Gamma^C (g, g')\}$.

Intuitively, we are finding the states from which an agent (or group of agents) cannot know ϕ (that is, there exists some global state where ϕ does not hold that is indistinguishable to the agent or group of agents), and then taking the complement.

2.3.6 Summary

There are a "whole zoo" [54] of temporal logics available that can capture varying properties of interest; in this section we have discussed the syntax, semantics, expressivity and complexity of four popular temporal specification languages, briefly mentioning existing model checking algorithms where appropriate. We have shown that some of them have incomparable expressivity (CTL and LTL; CTL* and LDL) in that in both directions, there are properties expressible in one logic but not in the other. Generally, additional expressivity tends to come at a cost in terms of computational complexity; further to that, especially if one considers industrial applications of verification the utility of what *can* be expressed is also an important consideration [95]. Furthermore, there are many cases where algorithms do not actually reach their worst-case complexity; in spite of LTL and CTL* model checking both being PSPACE-complete, the former has often been faster in practice [84].

We have also outlined observational semantics for epistemic modalities, and shown how they can augment LTL to allow us to reason about knowledge with a linear-time view using LTLK. Similar approaches can be used to extend the other logics with such capabilities [66].

2.4 Verification Techniques

In this section we discuss some of the methods by which specifications can be verified; that is, given an interpreted system $\mathcal{I}$ and modal formula ϕ, for a given global state g how we compute whether $\mathcal{I}, g \vDash \phi$. In practice, as previously discussed it is often more convenient to compute the set of states in which ϕ holds, $[\phi]_{\mathcal{I}}$, and then check whether $g \in [\phi]_{\mathcal{I}}$ [66].

2.4.1 Explicit Construction

One possible approach involves explicitly constructing a directed graph representing $\mathcal{I}$; this would involve constructing nodes for the global states G and edges depending on the evolution function t. Thereafter, we can attempt to directly evaluate properties on said graph; for example, for model checking CTL we can explicitly implement the state-labelling algorithms outlined in Section 2.3.2. However, this approach does not scale well; it is particularly vulnerable to the *state explosion problem*, in that the size of the model (here graph) will grow exponentially as additional components are introduced to a system [32]. Explicitly enumerating the states of the model does not work well with this, as many algorithms tend to require set operations over states that will be very costly (in general, linear in the size of the model *per set operation*).

In practice, explicit approaches are often only able to cope with models with order 10^6 states [66]. Through the use of more sophisticated methods of representing the model and states, it is possible to perform better in practice.

2.4.2 Binary Decision Diagrams

Binary decision diagrams can be used to represent Boolean functions in a compact manner. In general, for a Boolean function with n variables, many "classical" representations such as truth tables, Karnaugh maps or sum-of-products form have many or all functions requiring exponential space; more practical representations such as reduced sums of products still have weaknesses in that they lack canonicity and can result in sudden degradations in performance even when simple operations such as complementation are performed [23]. Binary decision diagrams attempt to address some of these weaknesses.

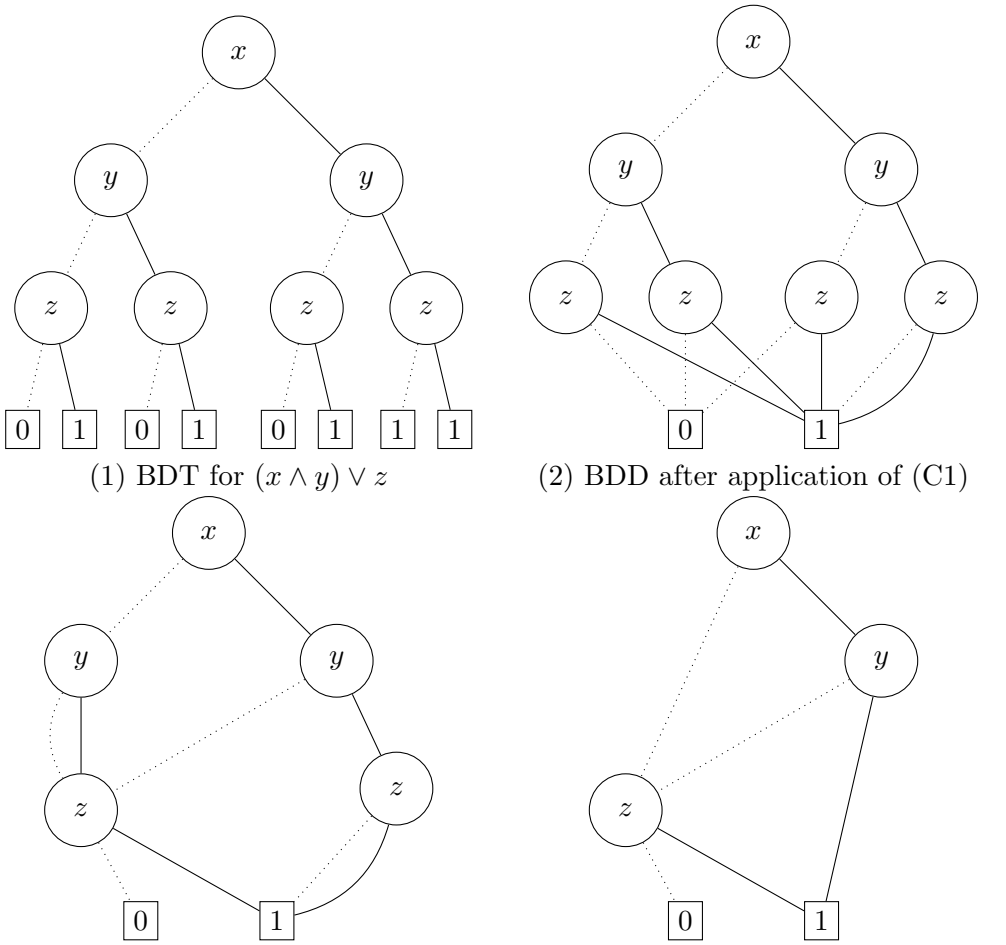


Figure 2.3: BDT for $(x \wedge y) \vee z$ and its reduction to a reduced BDD.

Definition 2.25. (Binary Decision Trees) A *binary decision tree* (BDT) is a complete binary tree with a unique starting node. Within the BDT, each internal node is labelled with a Boolean variable and each terminal node is labelled either '0' or '1'; all internal nodes on the same level of the tree share the same label. Each internal node has 2 outgoing edges, one labelled '0' and one labelled '1'. We adopt a convention that the '0' edge is represented by a dotted line and '1' by a solid line.

BDTs serve as a representation of a Boolean function $f(x_1, x_2 \ldots x_n)$. To evaluate $f(x_1, x_2 \ldots x_n)$, we proceed as follows: beginning from the starting node, we traverse the graph. When visiting an internal node labelled with variable x_i, we follow the outgoing '0' edge if $x_i = 0$ and the outgoing '1' edge if $x_i = 1$. When we reach a terminal node, the value of said terminal (either 0 or 1) is the value of f for the relevant values of x_i.

For example, consider the Boolean function $f(x, y, z) = (x \wedge y) \vee z$. We can construct a BDT for f by selecting an ordering of the variables and building a suitable tree. For example, consider Figure 2.3.

BDTs clearly have size exponential in the number of inputs to the Boolean function being

considered. Nonetheless, we can apply various heuristics to, in many cases, significantly reduce the size of a BDT [66].

- **(C1) Removal of duplicate terminals.** We can condense all terminal nodes to (up to) two nodes – one representing 0 and another representing 1. We can then redirect all edges pointing to 0 terminals to the one 0 and all edges pointing to 1 terminals to the one 1.
- **(C2) Removal of redundant tests.** If both outgoing edges of a node m point to the same node n, we can remove m from the BDD and redirect all edges incident on m to n.
- **(C3) Removal of duplicate non-terminals.** If two distinct nodes m and n are roots of structurally identical sub-BDDs, we can condense them into a single node (say n) and redirect all edges incident on m to n.

We now show concretely how these rules can be used to reduce the BDT in Figure 2.3 while preserving an equivalent structure:

- We first apply (C1), merging all of the terminals for 0 and 1 to a single terminal for each value. We obtain the BDD (2) in Figure 2.3.
- We then apply (C3); notice that all internal nodes labelled z apart from the rightmost one have their '0' outgoing edge going to the terminal 0 and their '1' outgoing edge going to the terminal 1. We now have the BDD (3) in Figure 2.3.
- We then apply (C2) twice: notice that the left internal node labelled y and the right internal node labelled z have both outgoing edges pointing to the same node, and can thus be removed. We now obtain the BDD (4) in Figure 2.3; no further simplifications are possible.

We can generalise BDTs to BDDs, as follows.

Definition 2.26. (Binary Decision Diagrams) A *binary decision diagram* (BDD) is a finite directed acyclic graph with a unique initial node. Within the BDD, each internal node is labelled with a Boolean variable and each terminal node is labelled either '0' or '1'. Each internal node has 2 outgoing edges, one labelled '0' and one labelled '1'. We adopt the same convention as we did for BDTs; the '0' edge is represented by a dotted line and the '1' edge by a solid line. The semantics of BDDs (in terms of the way they represent a Boolean function) are identical to that for BDTs.

Definition 2.27. (Reduced Binary Decision Diagrams) A binary decision diagram is said to be *reduced* if none of the optimisations C1 – C3 as defined earlier can be applied.

BDDs and reduced BDDs as defined in Definitions 2.26 and 2.27 are not canonical in that a formula can have multiple representations (even in the reduced case). For example, the reduced BDD (4) in Figure 2.3 and the reduced BDD in Figure 2.4 both represent the same boolean function $(x \wedge y) \vee z$. We now discuss how a canonical representation for a given formula can be obtained, through the use of *variable orderings*.

Definition 2.28. (Variable Ordering and Ordered BDDs) Let $L = [x_1, x_2, \ldots x_n]$ be an ordered list of variables without duplications, and let B be a BDD. B has the *variable ordering* L iff the label of every internal node is contained in L and, for every occurrence of x_i followed by x_j along a path in B we have $i < j$. An *ordered BDD* (OBDD) is a BDD which has an ordering for *some* list of variables.

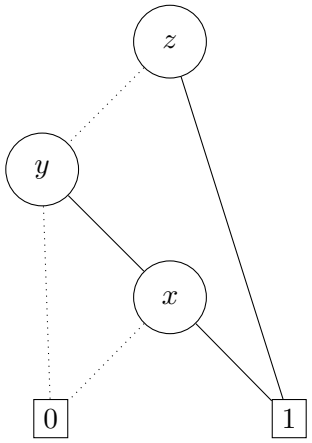


Figure 2.4: Reduced BDD for $(x \wedge y) \vee z$ using a different variable ordering.

Definition 2.29. (Compatible Variable Orderings) Let B_f and B_g be OBDDs with variable orderings L_f and L_g respectively. B_f and B_g have *compatible variable orderings* iff there does not exist a pair (x, y) such that x comes before y in L_f and y comes before x in L_g.

It is known that once a variable ordering has been determined, the reduced OBDD corresponding to a given Boolean function f is unique; furthermore, if two reduced OBDDs B_f and B_g have compatible variable orderings then they must have the same structure [66].

The selection of which variable ordering should be used to optimise the size of the reduced OBDDs is known to be an NP-complete problem [22]. Nonetheless, it is important to avoid worst-case performance; for example, for Boolean functions of the form $\phi_N = \bigvee_i (x_i \wedge y_i)$ for integer $0 \leq i < N$, optimal orderings such as $[x_0, y_0, x_1, y_1, \ldots, x_{N-1}, y_{N-1}]$ yield reduced OBDDs of linear size (here $2N+2$), while worst-case orderings such as $[x_0, x_1, \ldots, x_{N-1}, y_{N-1}, y_{N-2}, \ldots, y_0]$ yield reduced OBDDs of exponential size (here 2^{N+1}) [23]. In practice, many BDD packages such as CUDD [88] use the *sifting algorithm* which involves a dynamic local search heuristic; see [81] for more detail.

Another advantage of BDDs is that, in general, performing tests and comparisons as well as various operations on BDDs can be efficient. Supposing f, g are Boolean functions with the same variable ordering, checking if f and g are equivalent can be done by checking if their BDDs B_f and B_g are structurally identical. Checking validity or satisfiability for a given Boolean function f is even simpler; we simply check if the BDD B_f is equal to just the single terminal B_1 (valid iff equal), or if it is equal to the single terminal B_0 (satisfiable iff *not* equal) [66]. Several other operations such as complementation (constant time by swapping the 0 and 1 terminals) or finding a satisfying assignment can be done very efficiently as well, especially if the BDDs are small [86]. A more detailed discussion of how Boolean operations may be applied on BDDs can be found in [23].

2.4.3 Symbolic Model Checking

Symbolic model checking involves encoding sets (in particular, sets of states) and functions over sets as Boolean formulas, and manipulating these Boolean formulas to determine if $\mathcal{I}, g \vDash \phi$ for some interpreted system $\mathcal{I}$, global state g and specification ϕ. In particular, this can be done without explicitly enumerating the global states G of $\mathcal{I}$; we use the reduced OBDDs introduced

in Section 2.4.2 to perform these operations efficiently. The technique can often handle state spaces as large as 10^{25} states [66].

Suppose we have a set $S = \{a, b, c\}$. Then, we can identify an element in S using $n = 2$ bits (in general, we need $\lceil \log_2 |S| \rceil$ bits). For example, using two boolean variables x_1 and x_2, we can encode a as $\neg x_1 \wedge \neg x_2$, b as $\neg x_1 \wedge x_2$ and c as $x_1 \wedge x_2$, leaving $x_1 \wedge \neg x_2$ as an uninstantiated element [66].

Subsets of S can then be represented by Boolean formulas f using n variables. Suppose $X \subseteq S$. We can encode X by choosing $f_X = \bigvee_{x \in X} e(x)$ where $e(x)$ is the encoding of element x into a Boolean formula. For example, the subset $Y = \{a, b\}$ can be encoded as $f_Y = (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) = \neg x_1$. We can then represent f_Y (or in general the formula for any subset) with an OBDD after fixing a suitable variable ordering. To improve this encoding, we can decide to include uninstantiated elements in our encoding if it will improve the size of the reduced OBDDs; the entire set S can be represented as $f_S = (\neg x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2) \vee (x_1 \wedge x_2) = \neg(x_1 \wedge \neg x_2)$, though obviously $f_S = \top$ is also acceptable (as we don't care about whether $x_1 \wedge \neg x_2$ is present or not) and gives rise to a smaller reduced OBDD [66]. In particular, this allows us to represent subsets of the global states G of $\mathcal{I}$ using BDDs.

We can thus encode set operations as Boolean operations on their representative formulas; clearly, $f_{V \cap W} = f_V \wedge f_W$, $f_{V \cup W} = f_V \vee f_W$ and $f_{-V} = \neg f_V$. These operations can be performed symbolically on binary decision diagrams for efficiency.

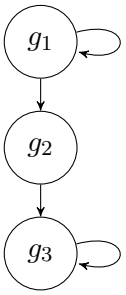
We proceed by constructing the transition relation of $\mathcal{I}$; that is, $R = \{(g, g') : \exists a \in Act. g' = t(g, a)\}$. Clearly, $R \subseteq G \times G$. If we encode two copies of G, one using the variables $x_1, \ldots x_n$ and another using $x'_1, \ldots x'_n$ then we can express R as a BDD over the two sets of variables[2]. In practice, it has been found that interleaving variables from the current and next states tends to lead to more efficient reduced OBDDs [66].

We illustrate this construction with a concrete example, illustrated in Figure 2.5. Consider a simple interpreted system with three global states; $G = \{g_1, g_2, g_3\}$ and a transition relation $R = \{(g_1, g_2), (g_2, g_3), (g_1, g_1), (g_3, g_3)\}$. We can choose an encoding for the states of G using two propositional variables x_1 and x_2; we represent g_1 by $\neg x_1 \wedge \neg x_2$, g_2 by $x_1 \wedge \neg x_2$ and g_3 by $x_1 \wedge x_2$. We can then encode the transition relation as in (3) in Figure 2.5 where '-' represents the "don't care" states (where we have either $\neg x_1 \wedge x_2$ or $\neg x'_1 \wedge x'_2$ since they reference an uninstantiated state), and then determine a suitable Boolean expression for the transition relation R, such as $f_R = (\neg x_1 \wedge \neg x'_2) \vee (x_1 \wedge x'_2)$. (Of course, other choices of f_R are possible, depending on which "don't care" states one includes in one's choices.).

Many of the model checking algorithms depend on our ability to compute universal and existential preimages (as defined in Definition 2.13). It may be noted that the universal image may be expressed in terms of an existential preimage, as follows:

$$\begin{aligned} \mathrm{pre}_\forall(X) &= \{g \in G : \forall g' \in t(g, a). g' \in X\} \\ &= G - \{g \in G : \neg \forall g' \in t(g, a). g' \in X\} \\ &= G - \{g \in G : \exists g' \in t(g, a). g' \notin X\} \end{aligned}$$

[2]It is certainly possible to use fewer variables if we attempt to take R as a direct subset of $G \times G$, but this adds significant complexity to the implementation. Furthermore, this can make taking the projection of one set of the global states, which is needed when taking the existential preimage, somewhat complicated.



(1) Transition relation

State	Encoding
g_1	$\neg x_1 \wedge \neg x_2$
g_2	$x_1 \wedge \neg x_2$
g_3	$x_1 \wedge x_2$

(2) State encoding

x_1	x_2	x_1'	x_2'	R
0	0	0	0	1
0	0	0	1	-
0	0	1	0	1
0	0	1	1	0
0	1	0	0	-
0	1	0	1	-
0	1	1	0	-
0	1	1	1	-
1	0	0	0	0
1	0	0	1	-
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	-
1	1	1	0	0
1	1	1	1	1

(3) Transition relation encoding

Figure 2.5: Encoding of a simple interpreted system.

$$= G - \{g \in G : \exists g' \in t(g,a).g' \in G - X\}$$
$$= G - \text{pre}_\exists(G - X)$$

Thus, it is sufficient to outline how to compute an existential preimage – recall that $\text{pre}_\exists(X) = \{g \in G : \exists a \in Act.t(g,a) = g' \wedge g' \in X\}$; equivalently we have $\text{pre}_\exists(X) = \{g \in G : R(g,g') \wedge g' \in X\}$. Suppose we have for a set of states X a Boolean function f_X and a BDD B_X, and for the transition relation R a Boolean function f_R and BDD B_R.

1. We first relabel every internal node in B_X to use the "primed" version of the variable. This gives us a BDD $B_{X'}$.

2. We use the *apply* algorithm as documented in [23] to compute a BDD representing $f_{X'} \wedge f_R$; that is, $\text{apply}(\wedge, B_R, B_{X'})$. The procedure is based on the Shannon expansion

$$f_1 < op > f_2 = ((\neg x_i) \wedge (f_1|_{x_i=0} < op > f_2|_{x_i=0})) \vee (x_i \wedge (f_1|_{x_i=1} < op > f_2|_{x_i=1}))$$

   Suppose we are trying to compute $\text{apply}(< op >, B_1, B_2)$ where B_1 and B_2 are BDDs with compatible variable orderings. Since they have compatible variable orderings, it is possible to topologically sort all of the variables in B_1 and B_2 to produce a combined ordering $[x_1, x_2, \ldots x_n]$. We proceed recursively as follows:

   - If both B_1 and B_2 are terminal nodes then $B_1 < op > B_2$ can be directly calculated and is a terminal node.
   - Otherwise, if B_1 and B_2 have their starting node labelled with the same variable x_i, then we return a BDD with starting node x_i. We recursively apply $< op >$ to the '0' children of both B_1 and B_2 to calculate the '0' child of the starting node in the output BDD, and use a similar method to calculate the '1' child.

- Otherwise, suppose that B_1 has a starting node with label x_i, and B_2 has a starting node that is a terminal or has label x_j where $j > i$. Then B_2 must be independent of i (since $j > i$ and we have topologically sorted the variable ordering). We can thus return a BDD with starting node x_i, and recursively apply $< op >$ to the '0' child of B_1 and (all of) B_2 to calculate the '0' child of the starting node in the output BDD, and use a similar method to calculate the '1' child.
- Otherwise, B_2 has a starting node with label x_j, and B_1 has a starting node that is a terminal or has label x_i with $i > j$. We proceed as in the third case, with B_1 and B_2, x_i and x_j swapped.

In practice, the above algorithm is implemented with a dynamic programming optimisation to avoid evaluating any given pair of subgraphs more than once [23]. This results in a BDD $B_{R \wedge X'}$; we need to extract the current states that are related to at least one state in X'.

3. We use the *exists* algorithm to return the set of states (defined over the non-primed variables) for which there exists at least one next state. We have $f_{pre_\exists(X)} = \exists x'_1 \exists x'_2 \ldots \exists x'_n (f_R \wedge f_{X'})$; to evaluate a single existential quantification we observe that $\exists x f = f|_{x=0} \vee f|_{x=1}$. Thus, for each variable x'_i we need to compute $\text{apply}(\vee, \text{restrict}(0, x_i, B_f), \text{restrict}(1, x_i, B_f))$ where the restrict function traverses the BDD; when encountering internal nodes labelled x_i, we delete them and replace all incident edges with the appropriate child of that node. This allows us to compute a BDD for $f_{pre_\exists(X)}$.

This completes our discussion of how BDDs may be used for symbolic model checking. These techniques are instrumental in allowing in our implementation of model-checking algorithms for LTLK, CTL*K and LDL to scale to larger state-spaces.

2.5 LTL Model Checking in Greater Detail

In this section we briefly discuss the automata-theoretic basis behind algorithms for model-checking LTL. We then outline in detail the *tableau construction* of Clarke et al. [28], involving the reduction of the LTL model checking problem to that of CTL model checking with fairness constraints, via the construction of a symbolic Büchi automaton.

2.5.1 Büchi Automata

One approach for model checking LTL specifications (and model checking in general) involves viewing evolution paths of systems as infinite words over some alphabet, as documented in [94].

Definition 2.30. (Automata) Let an *alphabet* Σ be a nonempty set. An *infinite word* is then an infinite sequence $a_0, a_1 \ldots$ of symbols in Σ. A nondeterministic *automaton* A is a tuple $A = (\Sigma, S, S^0, \rho, F)$ where Σ is an alphabet, S is a finite nonempty set of *states*, $S^0 \subseteq S$ is a finite nonempty set of *initial states*, $F \subseteq S$ is a set of *accepting states* and $\rho : (S \times \Sigma) \to 2^S$ is a (partial) *transition function.*

Definition 2.31. (Büchi Automata) Let $A = (\Sigma, S, S^0, \rho, F)$ be an automaton. Define a *run* r of A on an infinite word $w = a_0, a_1, \ldots$ as a sequence of states $s_0, s_1, \ldots$ where $s_0 \in S^0$ and $s_{i+1} = \rho(s_i, a_i)$ for every $i \geq 0$. Let $\lim(r)$ refer to the set of states visited infinitely often by r; this must be nonempty since r is infinite while S is not.

A is a *Büchi automaton* if we define r as an *accepting run* iff some accepting state is reached

infinitely often (formally if there exists $f \in F$ with $f \in \lim(r)$). An infinite word w is then *accepted* by A iff there exists some accepting run of A with input w. The *language* of A, $L_\omega(A)$ is the set of infinite words accepted by A.

Intuitively, if we consider $\Sigma = 2^{AP}$ (where AP is the set of atomic propositions), we can express paths as infinite words over Σ. In particular, it has been shown that given an LTL formula ϕ, one can construct a *Büchi automaton* which characterises *precisely* the evolution paths satisfying ϕ – formally, given a path π there exists a Büchi automaton A such that $\pi \vDash \phi$ iff $\pi \in L_\omega(A)$. The details of this construction are outlined in [79]; we do not go into further detail as we are not employing such a direct translation. Alternative constructions such as *alternating automata* [94] also exist.

Once we have constructed an automaton A_ϕ for our LTL formula ϕ, we can use it for model checking by constructing a Büchi automaton for the system $\mathcal{I}$ being modelled that accepts any valid runs; this can be done by defining $A_\mathcal{I} = (\Sigma, G, I, R, G)$ where $R = \{(g, g') : \exists a \in Act.g' = t(g, a)\}$. The model checking problem then reduces to verifying that all computations accepted by $A_\mathcal{I}$ are also accepted by A_ϕ; equivalently verifying *emptiness* of the automaton accepting $L_\omega(A_\mathcal{I}) \cap L_\omega(A_{\neg\phi})$ – this automaton can be constructed from the automata from $A_\mathcal{I}$ and $A_{\neg\phi}$ (e.g. using constructions detailed in [94]).

It is thus possible to perform LTL model checking by explicitly building the Büchi automaton (or building other automata that are able to characterise LTL specifications), and then composing it with the automaton representing our model – and this approach is used by several model checkers, such as SPIN [50] or SPOT [38]. However, these approaches tend not to scale as well as symbolic approaches [78].

2.5.2 Tableau Construction: Reduction to CTL Model Checking

It is possible to construct a symbolic Büchi automaton for an LTL formula, using what is known as the *tableau method* [28]. This approach is used by many modern symbolic model-checkers such as NuSMV and CadenceSMV [79], and these implementations have been found to scale better than many tools that use explicit constructions [78]. This can be used to reduce the problem of LTL model checking to CTL model checking with fairness constraints.

Observe that to carry out LTL model checking it suffices to, given an LTL formula ϕ, be able to determine if there *exists* a path satisfying ϕ. To determine LTL satisfaction over states, which depends on *all* paths from a given state satisfying ϕ, observe that

$$\begin{aligned}
\mathcal{I}, g \vDash \phi &\leftrightarrow \mathcal{I}, \pi \vDash \phi \text{ for all paths } \pi \text{ starting at } g \\
&\leftrightarrow \text{it is not the case that (not } (\mathcal{I}, \pi \vDash \phi \text{ for all paths } \pi \text{ starting at } g)) \\
&\leftrightarrow \text{it is not the case that } (\mathcal{I}, \pi \nvDash \phi \text{ for some path } \pi \text{ starting at } g) \\
&\leftrightarrow \text{it is not the case that } (\mathcal{I}, \pi \vDash \neg\phi \text{ for some path } \pi \text{ starting at } g)
\end{aligned}$$

Intuitively, this holds from the CTL* equivalence $Af = \neg E \neg f$; the ability to determine if there *exists* a path with an arbitrary LTL specification holding gives us the facility to find the states in which an arbitrary LTL specification ϕ holds, by finding the states on which there exists a path with $\neg\phi$ and taking the complement. We thus focus on, given an interpreted system $\mathcal{I}$, global state g and LTL specification ϕ, determining if there exists a path π starting at g such that $\mathcal{I}, \pi \vDash \phi$. At a high level, the algorithm works as follows, adapting the construction from [28] to work over interpreted systems:

- Given ϕ, we construct a Kripke structure for ϕ called the *tableau* T, encoding all of the paths satisfying ϕ.
- Compose T and $\mathcal{I}$, finding the set of paths appearing both in T and $\mathcal{I}$. A global state in $\mathcal{I}$, say g, has a path satisfying ϕ if and only if there is a path in the composition satisfying ϕ starting from g.

We first outline the tableau construction for a given LTL formula ϕ. Note that this construction is independent of the system $\mathcal{I}$ under consideration.

Definition 2.32. (Elementary Formulas) For a given LTL formula ϕ, the *elementary formulas* of ϕ, $el(\phi)$ are recursively defined, as follows:

- $el(p) = \{p\}$ (where p is an atomic proposition).
- $el(\top) = \emptyset$.
- $el(\neg\phi) = el(\phi)$.
- $el(\phi \wedge \psi) = el(\phi) \cup el(\psi)$.
- $el(X\phi) = \{X\phi\} \cup el(\phi)$.
- $el(\phi U \psi) = \{X(\phi U \psi)\} \cup el(\phi) \cup el(\psi)$.

The set of states in the tableau for ϕ, S_T is given by the power-set of $el(\phi)$, $\mathcal{P}(el(\phi))$. The truth assignment for atomic propositions in the tableau, h_T is as follows: p holds in precisely the states containing p. We also define a function *sat*, which associates each subformula of the original formula ϕ with a set of states in the tableau.

Definition 2.33. (*sat* function) The *sat* function from subformulas ψ of ϕ to states of the tableau (hence sat : (subformulas of ϕ) $\rightarrow \mathcal{P}(el(\phi))$) is defined as follows:

- $sat(\psi) = \{\sigma : \psi \in \sigma\}$ (for ψ being an elementary formula).
- $sat(\top) = S_T$.
- $sat(\neg\psi) = S_T - sat(\psi)$.
- $sat(\psi_1 \wedge \psi_2) = sat(\psi_1) \cap sat(\psi_2)$.
- $sat(\psi_1 U \psi_2) = sat(\psi_2) \cup (sat(\psi_1) \cap sat(X(\psi_1 U \psi_2)))$.

Note that we do not need to separately handle subformulae of the form $X\psi$ as these are necessarily elementary formulas based on Definition 2.32 and thus fall under the first case.

Intuitively, $sat(\psi)$ captures the states in the tableau in which ψ is satisfied. We can then define the transition relation for the tableau, which focuses on ensuring that the semantics of elementary formulas of the form $X\psi$ are respected.

Definition 2.34. (LTL Tableau Transition Relation) Suppose R_T is the transition relation for the tableau T for formula ϕ. This is defined by

$$R_T(\sigma, \sigma') = \bigwedge_{(X\psi) \in el(\phi)} \big(\sigma \in sat(X\psi) \leftrightarrow \sigma' \in sat(\psi)\big)$$

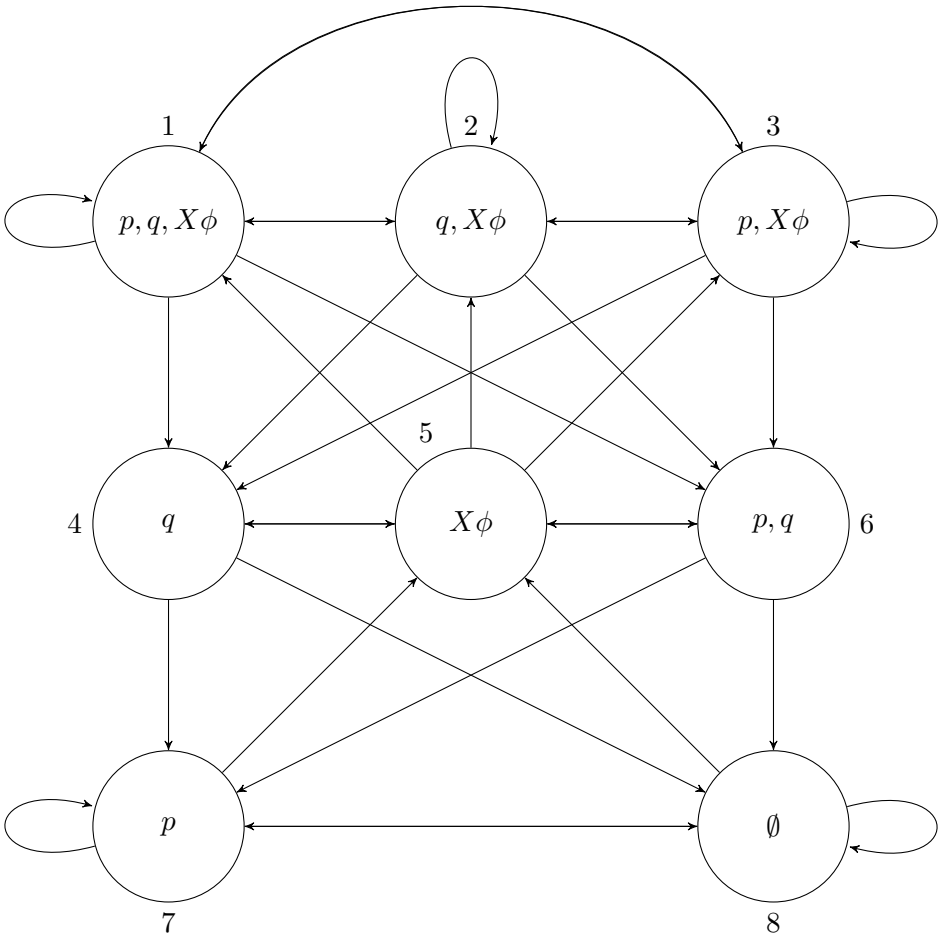


Figure 2.6: Tableau for pUq.

We use an example from [28]. Consider the formula $\phi = pUq$. Clearly the elementary formulas of ϕ are simply $el(\phi) = \{p, q, X\phi\}$. There are thus 8 states in the tableau corresponding to the 8 possible subsets of $el(\phi)$.

Consider the labelling in Figure 2.6. Clearly, $sat(X\phi) = \{1, 2, 3, 5\}$ since those are the states with $X\phi$. $sat(\phi) = sat(pUq) = sat(q) \cup (sat(p) \cap sat(X\phi)) = \{1, 2, 3, 4, 6\}$. Thus, we have transitions from each state in $sat(X\phi)$ to each state in $sat(\phi)$, and transitions from each state in $sat(\neg X\phi) = S_T - sat(X\phi)$ to each state in $sat(\neg\phi) = S_T - sat(\phi)$.

Notice that the definition of R_T does not guarantee *eventuality* properties, so it is not necessarily the case that all paths beginning at a state $\sigma \in sat(\phi)$ satisfy ϕ [28]. For example, consider the path $\pi = 3^\omega$; while state 3 is in $sat(pUq)$, pUq clearly does not hold on π as q never becomes true. We thus need to add an additional condition: for every subformula of the form $\psi_1 U \psi_2$ in ϕ and for every state σ on π, if $\sigma \in sat(\psi_1 U \psi_2)$ and $\sigma \notin sat(\psi_2)$, then there must exist a later state on π in $sat(\psi_2)$. In other words, ψ_2 must hold at some future point. This is implemented by adding a fairness constraint $\neg(\psi_1 U \psi_2) \vee \psi_2$ for every such subformula. Intuitively, T now includes every path satisfying ϕ; if $\mathcal{I}, \pi \vDash \phi$ then there must be a path π' in T starting at some state in $sat(\phi)$ with the atomic propositions in π' and π being consistent.

Please see [28] for a full proof.

We then need to determine the *product* of the tableau T and our interpreted system $\mathcal{I}$. Here, we adapt the definition of product from [28]:

Definition 2.35. (Product of Tableau and Interpreted System)
Let $T = (S_T, R_T, h_T)$ be the tableau as constructed using Definitions 2.32 and 2.34, and $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h_{\mathcal{I}} \rangle$ be an interpreted system. The *product* of T and $\mathcal{I}$ may be described as a Kripke structure (S, R, h) as follows.

- $S = \{(\sigma, g) : \sigma \in S_T \wedge g \in G \wedge \forall p \in AP. \sigma \in h_T(p) \leftrightarrow g \in h_{\mathcal{I}}(p)\}$.
- $R = \{((\sigma_1, g_1), (\sigma_2, g_2)) : R_T(\sigma_1, \sigma_2) \wedge \exists a \in Act. t(g_1, a) = g_2\}$.
- $h(p) = \{(\sigma, g) : \sigma \in h_T(p)\}$ for every $p \in AP$.

We also extend the definition of *sat* over S; $sat(\sigma, g)$ holds iff $sat(\sigma)$.

Intuitively, P contains the sequences for which there are paths in both T and M which are consistent in their atomic propositions. We thus need to find the states in the product that are in $sat(\phi)$ that satisfy all of the eventuality constraints. More formally, as described in [28], we have that $\mathcal{I}$ has a path starting at g satisfying ϕ iff there is some $\sigma \in S_T$ such that $(\sigma, g) \in sat(\phi)$ and $P, (\sigma, g) \vDash EG\top$ with the fairness constraints $\neg(\psi_1 U \psi_2) \vee \psi_2$ for every subformula $\psi_1 U \psi_2$ of ϕ.

All of these procedures can be implemented symbolically using BDDs – please consult Section 3.2 for implementation details.

2.5.3 Counterexample Generation

A useful feature of model checkers is their ability to return *counterexample* or *witness* traces, explaining why specifications are not satisfied or satisfied, respectively [31]. Since we demonstrated a reduction of the LTL model checking problem to the CTL model checking problem with fairness constraints in Section 2.5.2, we can use algorithms for finding witnesses to the CTL formula $EG\top$ with fairness constraints, such as in [29], to find counterexamples to an LTL formula ϕ. This involves first building the tableau for ϕ and composing it with the model $\mathcal{I}$, and then running our CTL with fairness constraints algorithm on the composition.

We now outline at a high level the algorithm from [29]. Suppose we want to find a witness trace for the formula $EG\phi$ with $\psi_1, \psi_2, \ldots, \psi_n$ as fairness constraints. Let f be the function

$$f(x) = [\phi]_{\mathcal{I}} \wedge \bigwedge_{i=1}^{n} [EX(E([\phi]_{\mathcal{I}} U (x \wedge [\psi_i]_{\mathcal{I}})))]_{\mathcal{I}}$$

Then, the greatest fixed point of f is the set of states satisfying $EG\phi$ with the fairness constraints $\psi_1, \ldots, \psi_n$, as shown in [29]. We proceed by attempting to compute a witness by connecting the *strongly connected components* of the state transition graph that satisfy each fairness constraint while ensuring that ϕ is always maintained throughout.

Definition 2.36. (Strong Connection, Strongly Connected Components) Suppose s, t are two nodes in a directed graph G. Then s and t are strongly connected iff there is a path from s to t and a path from t to s. Clearly, strong connection is an equivalence relation; we define the equivalence classes given by partitioning the nodes of G by this relation as the *strongly connected components* (SCCs) of G.

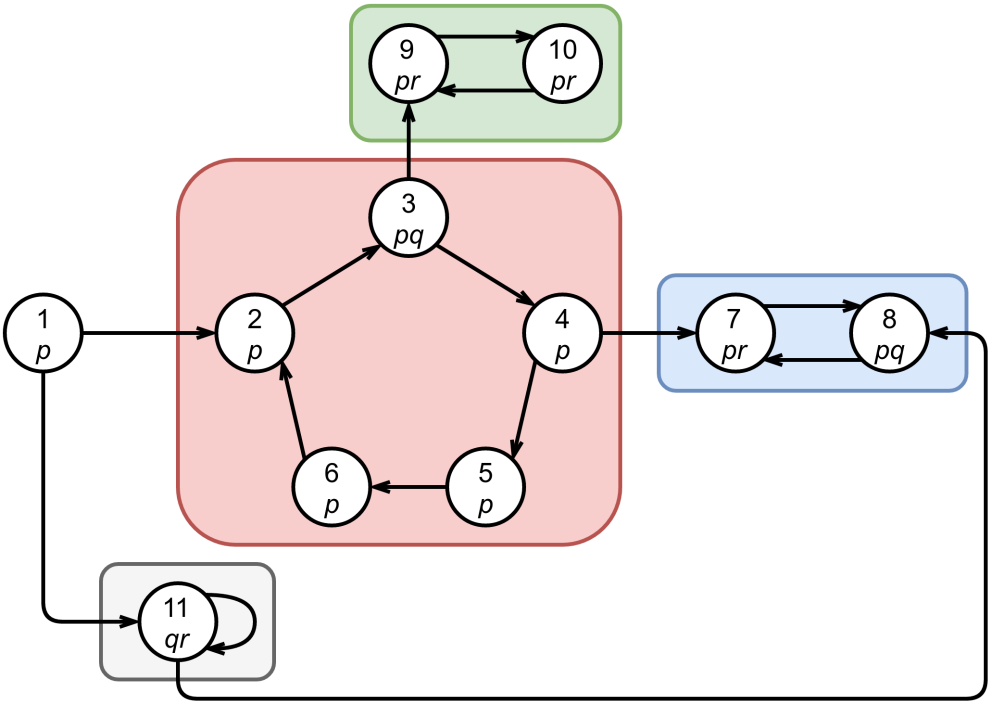


Figure 2.7: Sample state space over which we try to find a witness for EGp with fairness constraints q and r; each atomic proposition is true in precisely the states indicated with said proposition. Suppose state 1 is the starting state.

Throughout this section let us define the set of states satisfying $EG\phi$ as the set of states satisfying it under the fairness constraints $\psi_1, \ldots, \psi_n$. We illustrate the algorithm with an example – suppose we want to find a witness for EGp under fairness constraints q and r for the model in Figure 2.7.

We first evaluate the fix-point calculation of $E([\phi]_{\mathcal{I}} U(S \wedge [\psi_i]_{\mathcal{I}}))$, where S is the set of states in which $EG\phi$ holds under the fairness constraints. We store the intermediate results obtained from the fixed point calculation of $E([\phi]_{\mathcal{I}} U(S \wedge [\psi_i]_{\mathcal{I}}))$ – we define the intermediate result $Q_j^{\psi_i}$ as the set of states from which a state in $S \wedge [\psi_i]_{\mathcal{I}}$ is reachable in j steps while preserving ϕ [29]. For example, for the model above we have:

- $Q_0^q = \{3, 8\}$. We have $S = \{1, \ldots, 8\}$ since from these states (and only these states) we can have p until we transition into the blue SCC in Figure 2.7, which has both q and r infinitely often. Also, we have $[q]_{\mathcal{I}} = \{3, 8, 11\}$.
- $Q_1^q = \{2, 3, 7, 8\}$. 2 and 7 are successors of the previous set Q_0^q which satisfy p. Notice that we do not include 11 since it does not satisfy p.
- $Q_2^q = \{1, 2, 3, 4, 7, 8\}$.
- $Q_0^r = \{7\}$. This does not include 9, 10 or 11 because they are not in S.
- $Q_1^r = \{4, 7, 8\}$.
- $Q_4^r = \{1, 2, 3, 4, 7, 8\}$, by iteratively taking predecessors of the previous set which have p.

We then select some starting state s which satisfies $EG\phi$. For our model, the only starting state is 1, and it indeed satisfies EGp, so we select it. We attempt to greedily minimise the length of the witness path by selecting a fairness constraint ψ_i that can be reached in the smallest number of steps k (note that the problem of finding a witness of minimal length is NP-complete, as shown in [29]). This is clearly q, as it is reachable in 2 steps (while r is only reachable in 4).

We now determine a path to an actual state that satisfies $EG\phi \wedge \psi_i$; this can be done by iterating downwards through the $Q_j^{\psi_i}$ sets, from $j = k$ to 0, and choosing successors of the current node. For example, starting from state 1, we choose a successor of 1 in Q_1^q; we can choose 2. Then, we choose a successor of 2 in Q_0^q; we can choose 3. We thus have a path $1, 2, 3$. Let the actual state we finished in be t.

We then repeat the above process for the remaining fairness constraints, starting from t. For the example above, we only have r left, and we would construct the path $3, 4, 7$.

Once all fairness constraints have been considered, suppose we are in a state t_F. We then need to find a simple path from t_F back to our initial state t that has ϕ true on every state; this would complete a cycle that passes through every fairness constraint. We can thus obtain a witness for $EG\phi$, which would be the path that follows this cycle infinitely.

However, if finding such a path is not possible (and it is not possible in our example, since our starting state 1 is not a successor of any other state), we have from the definition of Q that t_F must also satisfy $EG\phi$. It suffices to restart the entire process from t_F [29]; t_F and t are not in the same SCC of the transition graph. Since t_F itself satisfies $EG\phi$, we can effectively eliminate all SCCs that cannot be reached from the SCC of t_F (as we can find a witness in the SCCs reachable from this SCC). In our example, we would restart from 7, and find that we already have r, and can obtain q by following the path $7, 8$. After that, we can indeed return to our starting state by the path $8, 7$.

The algorithm will terminate: the number of SCCs that we will need to consider has thus been reduced by at least one, since the SCC containing t is unreachable. Furthermore, since we are considering finite Kripke models, we can be sure the algorithm will eventually find a cycle (consider that once only one SCC needs to be considered, we will find a cycle since our starting point at that time will still satisfy $EG\phi$). We then return the concatenation of the path used to travel from s to t_F and the witness path found starting from t_F itself – for our example, this would be $1, 2, 3, 4, (7, 8)^\omega$.

This algorithm was implemented in MCMAS, though there were some issues with its implementation. We discovered a subtle bug in the existing implementation concerning the aforementioned example in that MCMAS would terminate instead of restarting if it was not possible to return to the starting state after satisfying all fairness constraints. There were also challenges in reusing it for LTLK model checking. Please consult Section 3.2.5 for more details.

2.6 LDL Model Checking in Greater Detail

In this section we discuss existing literature that is relevant to our LDL model checking algorithms. We first introduce Thompson's construction, which converts a regular expression into a nondeterministic finite automaton. We then introduce the concept of *alternating automata*, and then discuss an alternating automaton construction for LDL specifications introduced in [41].

We also introduce the *breakpoint construction* of [73], which can be used to reduce alternating automata to nondeterministic Büchi automata. We employ both of these techniques to perform model checking of LDL (and, as it turns out, CDL*) specifications – this is discussed in greater detail in Section 5.1.

2.6.1 ϵ-NFAs and Thompson's Construction

We first introduce automata with ϵ-transitions, and the concept of the ϵ-closure of a state (which is required to define acceptance in an ϵ-NFA). The definition is adapted from [52].

Definition 2.37. (ϵ-NFAs) An ϵ-NFA is a nondeterministic finite automaton which also features ϵ-transitions, which do not consume input symbols. Formally, an ϵ-NFA is a tuple $A = (\Sigma, S, s^0, \rho, f)$ where Σ is an alphabet, S is a finite nonempty set of *states*, s^0 is an *initial state*, f is an *accepting state* and $\rho : (S \times (\Sigma \cup \{\epsilon\})) \to 2^S$ is a (partial) *transition function*. Notice that we have restricted the definition of a general automaton, requiring $|S^0| = 1$ and $|F| = 1$ – the ϵ-NFAs generated by Thompson's construction follow this property.

Definition 2.38. (ϵ-Closure) Let s be a state in an ϵ-NFA $A = (\Sigma, S, s^0, \rho, f)$. The ϵ-closure of s, $E(s)$ refers to the set of states reachable from s by following zero or more ϵ-transitions; more formally, it is the minimum set of states $E(s)$ such that:

- $s \in E(s)$, and
- $t \in E(s) \wedge u \in \rho(t, \epsilon) \to u \in E(s)$.

Furthermore, for two states $s, t \in S$, we define $\Pi(s, t)$ as the set of all ϵ-paths from s to t – that is, all possible sequences $s_0, s_1, \ldots, s_n$ where $s_0 = s$, $s_n = t$, and $\rho(s_i, \epsilon) = s_{i-1}$ for $i \in \{1, \ldots, n\}$.

Definition 2.39. (Acceptance of ϵ-NFAs) Let $A = (\Sigma, S, s^0, \rho, f)$ be an ϵ-NFA. Then, we define a new relation $\rho' : (S \times \Sigma^*) \to 2^S$, which reflects the possible states one can transition to given each input word, as follows:

- $\rho'(s, \epsilon) = E(s)$ for every $s \in S$, and
- $\rho'(s, aw) = \bigcup_{t \in E(s)} \bigcup_{u \in \rho(t,a)} \rho'(u, w)$ for every $s \in S$, $a \in \Sigma$, $w \in \Sigma^*$.

We say that a word $w \in \Sigma^*$ is accepted by A if $f \in \rho'(s^0, w)$.

Thompson's construction is an algorithm for converting regular expressions to ϵ-NFAs, with languages equal to the set of words accepted by said regular expressions [89]. These nondeterministic finite automata feature ϵ-transitions, which may be followed without consuming a symbol from the input (in the context of model checking, this means that such transitions do not involve a transition in the underlying model). Apart from the test constructions in LDL, we can inductively construct ϵ-NFAs for the path expressions within the dynamic modalities of LDL using Thompson's construction, as follows:

- A propositional formula ϕ may be viewed as a transition from one state to another, which may be followed only if the current symbol satisfies ϕ. We can construct this as a two-state ϵ-NFA, with the state after the transition being accepting.
- For $\rho_1; \rho_2$, we intuitively need to match ρ_1 *and then* ρ_2. An ϵ-NFA for $\rho_1; \rho_2$ may be constructed as follows: Construct a starting state s^0, and add a single ϵ-transition into the initial state of ρ_1. Then, from the accepting state of the ϵ-NFA for ρ_1, add an ϵ-transition into the initial state of ρ_2. We add an ϵ-transition from the accepting state of ρ_2 into a new acecpting state.

- For $\rho_1 + \rho_2$, we intuitively need to accept every word accepted by either ρ_1 or ρ_2. An ϵ-NFA for $\rho_1 + \rho_2$ may be constructed by creating a new start state s^0 and then nondeterministically transitioning into the start state of the ϵ-NFAs for either ρ_1 or ρ_2. We need to accept a word if we end in either of the accepting states, so we construct a new final accepting state s' and add an ϵ-transition from the accepting states of both sub-ϵ-NFAs into s'.
- For ρ^*, we need to have the option of skipping the path expression altogether, as well as returning back to the start after a successful match. We thus construct the ϵ-NFA as follows: introduce a new initial state s^0 as well as a new final accepting state s'. We also add several ϵ-transitions that capture the aforementioned intuition:
  - from s^0 to s' (since zero iterations of ρ is acceptable),
  - from s^0 to the initial state of the ϵ-NFA for ρ,
  - from the accepting state of the ϵ-NFA for ρ to s', and
  - from the accepting state of the ϵ-NFA for ρ to its initial state.

The above constructions are graphically depicted in Figure 2.8.

2.6.2 Alternating Automata

LDL allows us to require that paths satisfy conjunctive properties. This is because it allows conjunction, and also allows use of tests. For example, consider the LDL formula $\langle \phi? \rangle \psi$. This formula is true on the paths that, from their start states, satisfy both ϕ and ψ; intuitively, to check if some path π satisfies the formula, we need to check that it satisfies ϕ and ψ.

If we use an automata-theoretic approach for model checking LDL specifications, to check a path π we would need to construct automata for both ϕ and ψ, and check that these automata both accept π. More generally, we need to extend the nondeterministic automata introduced in Section 2.5.1 to allow for simultaneous transitions into *multiple* states. This is feasible in the formalism of *alternating automata* [93]. We first define the set of *positive Boolean formulae* and what it means for a set of propositions to satisfy such formulae:

Definition 2.40. (Positive Boolean Formulae) Let AP be a set of atomic propositions. The set of *positive* Boolean formulas over AP, $\mathcal{B}^+(AP)$ refer to the Boolean formulas that can be constructed from the propositions of AP and the Boolean primitives $\top$ and $\bot$ using only $\wedge$ and $\vee$. Furthermore, we say that for $S \subseteq AP$ and Boolean formula $B \in \mathcal{B}^+(AP)$, S satisfies B if B is true when the propositions $p \in S$ are true and all other propositions are false.

We now formally define alternating automata.

Definition 2.41. (Alternating Automata) An *alternating automaton* A is a tuple $A = (\Sigma, S, S^0, \rho, F)$ where Σ is an alphabet, S is a finite nonempty set of *states*, $S^0 \subseteq S$ is a finite nonempty set of *initial states*, $F \subseteq S$ is a set of *accepting states* and $\rho : (S \times \Sigma) \to \mathcal{B}^+(S)$ is a (partial) *transition function*.

Intuitively, the boolean formulae $\mathcal{B}^+(S)$ reflect the possible *sets* of states we can transition into from the current state. For example, suppose we have $\rho(s_0, p) = (s_1 \vee s_2) \wedge (s_3 \vee s_4)$; this means that the automaton from state s_0, after processing the symbol p, transitions into one of the states from s_1 and s_2, **and** one of the states from s_3 and s_4. Due to the possibility of transitioning into multiple states after processing a symbol, runs of alternating automata are

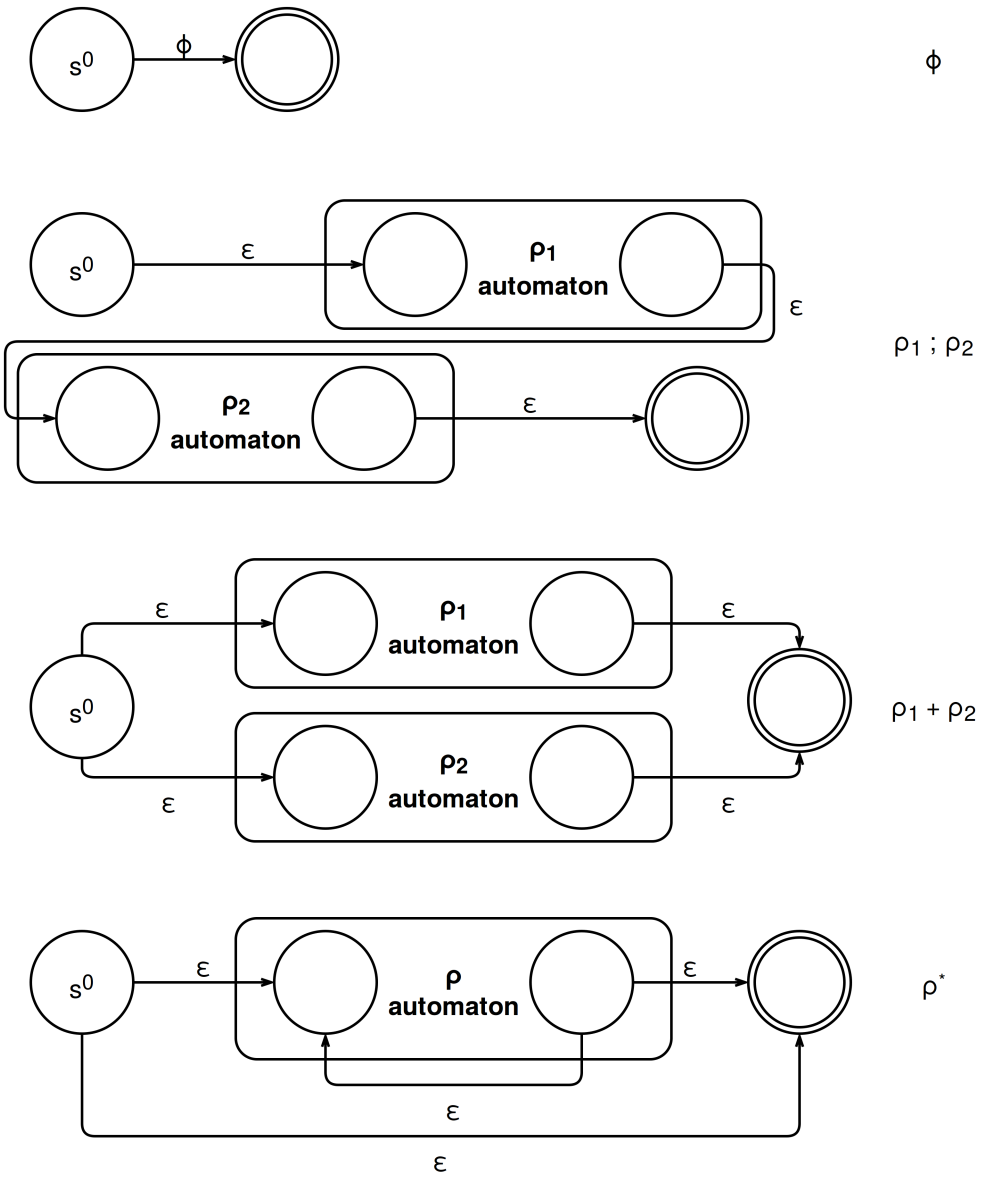
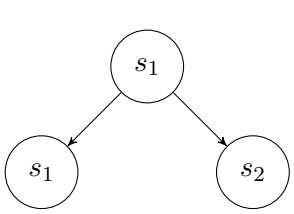
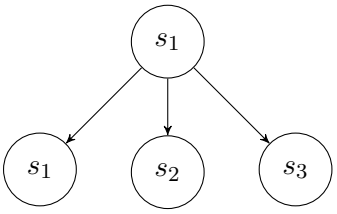


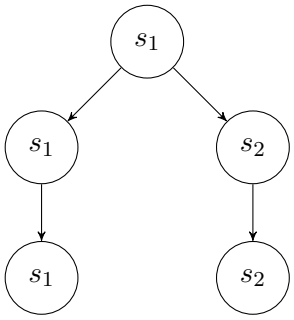
Figure 2.8: Thompson's construction for ϵ-NFAs. The blocks indicating sub-automata have their starting state on the left and accepting state on the right. Overall starting states are marked with s^0, and overall accepting states are depicted with a double circle.



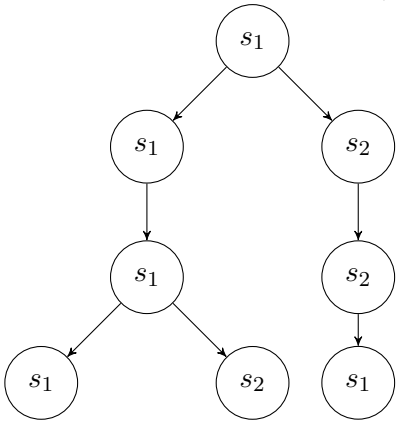
(1) Partial run tree of A (depth 1)



(2) Alternative partial run tree of A (depth 1)



(3) Partial run tree of A (depth 2)



(4) Partial run tree of A (depth 3)

Figure 2.9: Various run trees for the example alternating Büchi automaton A on $(ab)^\omega$.

trees rather than sequences [93].

Finally, we define alternating Büchi automata.

Definition 2.42. (Alternating Büchi Automata) Let $A = (\Sigma, S, S^0, \rho, F)$ be an alternating automaton. A *run* of A on some infinite word $w = a_0 a_1 \dots$ is a tree rooted at some $s^0 \in S^0$ such that, adapting the definition from [93]:

> If a node in the run tree s has distance i from the root of the tree, and $\rho(s, a_i) = \theta$, then x has k child subtrees $x_1, \dots, x_k$ for some $k \leq |S|$, and the set corresponding to the roots of these child subtrees satisfies θ.

Further to the above, we define a branch β in T as a sequence of nodes $x_0, x_1 \dots$ from the root where for every $i > 0$, x_i is a child of x_{i-1} in T. We define $\lim(\beta)$ as the set of states visited infinitely often in β. A is an *alternating Büchi automaton* if we define a run tree T to be accepting if for every branch β in T, we reach a transition to $\top$ or there exists some $f \in F$ where $f \in \lim(\beta)$.

Similar to nondeterministic Büchi automata (Definition 2.31), an infinite word w is then *accepted* by A iff there exists some accepting run of A with input w. The *language* of A, $L_\omega(A)$ is the set of infinite words accepted by A.

We illustrate the concept with an example. Consider the following alternating Büchi automaton $A = (\Sigma, S, S^0, \rho, F)$:

- $\Sigma = \{a, b\}$, $S = \{s_1, s_2, s_3\}$, $S^0 = \{s_1\}$, $F = \{s_1\}$, and
- $\rho(s_i, a) = \begin{cases} s_1 \wedge s_2 & i = 1, 3 \\ s_1 \vee s_3 & i = 2 \end{cases}$; $\rho(s_i, b) = s_i$ for all i;

Then, consider that since S^0 only contains s_1, all run trees must have s_1 at their root. Consider a run on the infinite word $(ab)^\omega$:

- Since $\rho(s_1, a) = s_1 \wedge s_2$, the children of the root node must contain s_1 and s_2. s_3 may or may not be a child of the root, so both run trees (1) and (2) in Figure 2.9 are permissible.
- Considering run tree (1), the next symbol of the word is b, and so the children of each state must include itself – so (3) in Figure 2.9 is a possible partial run tree.
- Next, observe that the next symbol of the word is now a, and we can expand the nodes for s_1 and s_2 following the transition relation. A possible expansion is (4) in Figure 2.9.
- Notice that we can always choose to transition from s_1 to **only** s_1 and s_2, and s_2 to only s_1 if the current symbol is a, and if it is b we can choose to transition from s_1 and s_2 to themselves only. Thus, it is possible to generate a run tree on which every branch only has s_1 and s_2. Furthermore, we cannot stay in s_2 infinitely, since we have a infinitely often and these necessitate a transition out of s_2. Hence, every branch has s_1 infinitely often, and thus there exists an accepting run of A on $(ab)^\omega$. Hence we say that A accepts $(ab)^\omega$.
- If we have $F = \{s_3\}$, observe that any run tree over $(ab)^\omega$ has a branch on which the *only* state is s_1, so our automaton has no accepting run; it does not accept $(ab)^\omega$ in this case.

2.6.3 LDL Specifications as Alternating Automata

We now introduce a translation of LDL formulae ϕ to suitable alternating Büchi automata A_ϕ, such that A_ϕ has an accepting run if and only if there exists a path satisfying ϕ. This construction was introduced in [41]; we adapt it for consistency with the notation we are using.

We first introduce a *negation normal form* for LDL.

Definition 2.43. (LDL Negation Normal Form) Let ϕ be an LDL formula. ϕ is in *negation normal form* if all negations apply *only* to atomic propositions, and it does not contain any instances of $\rightarrow$ or $\leftrightarrow$ (notice that these are alternative ways of expressing negation).

Converting an arbitrary LDL formula to negation normal form can be done by "pushing" in negations, using De Morgan's laws and the dual definitions of $\langle\rho\rangle\phi$ and $[\rho]\phi$:

- $\neg(\neg\phi) = \phi$
- $\neg(\phi \wedge \psi) = (\neg\phi) \vee (\neg\psi)$
- $\neg(\phi \vee \psi) = (\neg\phi) \wedge (\neg\psi)$
- $\neg(\phi \rightarrow \psi) = \neg(\neg\phi \vee \psi) = \phi \wedge (\neg\psi)$
- $\neg(\langle\rho\rangle\phi) = [\rho]\neg\phi$
- $\neg([\rho]\phi) = \langle\rho\rangle\neg\phi$

We now present the construction, assuming that the formula to be translated is in negation normal form. We first handle the propositional cases:

- For an atomic proposition ϕ, we transition into an accepting state if ϕ holds, and into a rejecting state if ϕ does not hold. This can be done with just one state, which transitions into $\top$ if ϕ holds, and into $\bot$ if ϕ does not hold.

- $\neg\phi$ is handled in the same way as above, except we transition into $\top$ if ϕ *does not* hold and $\bot$ if ϕ holds. (Recall that negation applies to atomic propositions only because we have assumed that the formula to be translated is in negation normal form.)
- If there exists a path satisfying $\phi \wedge \psi$, it must satisfy both ϕ and ψ. This may be handled by introducing a new initial state s^0, which has the conjunction of the transition relations of the original starting states. The overall automaton has accepting states equal to the union of the accepting states of the sub-automata.
- Similarly, if there exists a path satisfying $\phi \vee \psi$ it must satisfy *either* ϕ or ψ. This case can be handled by introducing a new initial state s^0, which has the union of the transition relations of the original starting states (for $\vee$). As before, the overall automaton has accepting states equal to the union of the accepting states of the sub-automata.

It remains to handle the modal cases. Recall that using Thompson's construction (introduced in Section 2.6.1) we can convert a regular expression into an ϵ-NFA. We can expand Thompson's construction to deal with tests, through the use of *marked ϵ-NFAs* from [41], as follows:

Definition 2.44. (Marked ϵ-NFA) A marked ϵ-NFA is an ϵ-NFA augmented with a marking function. Formally, a marked ϵ-NFA is a tuple $A = (\Sigma, S, s^0, \rho, f, m)$ where Σ is an alphabet, S is a finite nonempty set of *states*, s^0 is an *initial state*, f is an *accepting state*, $\rho : (S \times (\Sigma \cup \{\epsilon\})) \to 2^S$ is a (partial) *transition function* and $m : S \to \phi_{\text{LDL}}$ is a (partial) *marking function* assigning LDL formulae to states. Further to the above, if $\pi = s_0 \dots s_n$ is a path in a marked ϵ-NFA, we define the markings over π, $m(\pi) = \{m(s_i) \mid i \in \{0, \dots, n\}\}$.

Upon encountering a test $\theta?$, we convert it into a single-state automaton that is accepting, **but** is marked with the test θ; when combining sub-automata (e.g. for + and ; constructions) we preserve the existing markings. Intuitively, when a transition involves a test θ, we need to check that the test holds – a transition via *both* the starting state of the automaton and our original path in the ϵ-NFA of the regular expression captures this intuition.

We now introduce the construction for the diamond modality. Let $\langle\phi\rangle\psi$ be an LDL formula, where ϕ contains k tests $\theta_1, \dots, \theta_k$. We have

$$A_{\langle\phi\rangle\psi} = (\Sigma, S_r \cup S_\psi \cup S_1 \cup \dots \cup S_k, \{s_r^0\}, \rho, F_\psi \cup F_1 \cup \dots \cup F_k)$$

where S_r refers to the states of the marked ϵ-NFA corresponding to the modality ϕ, S_ψ the states of the automaton for ψ, and S_1 through S_k the states of the automata for the k tests; F_ψ refers to the accepting states of the automaton for ψ, and F_1 through F_k refer to the accepting states of the automata for the k tests. Furthermore, we have

$$\rho(s, A) = \begin{cases} \rho_\psi(s, A) & s \in S_\psi \\ \rho_j(s, A) & s \in S_j, j \in \{1, \dots, k\} \\ \bigvee_{s' \in S_r \setminus \{f_r\}} \bigvee_{\pi \in \Pi(s,s')} \bigvee_{t \in \rho_r(s', A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right) & \\ \vee & s \in S_r \\ \bigvee_{\pi \in \Pi(s, f_r)} \left(\rho_\psi(s_\psi^0, A) \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right) & \end{cases} \tag{2.1}$$

where $\Pi(s, t)$ refers to the set of ϵ-paths from s to t, as defined in Definition 2.38.

Intuitively, this construction is the union of the automata for the regular expression ϕ, its k

tests $\theta_1 \dots \theta_k$ and the following formula ψ, with the transition relation modified (in particular, we are adding transitions for the tests, as well as removing ϵ-transitions which are not allowed in alternating automata [41]). The transition relations for ψ and the tests are unaffected; the transition relation for the regular expression ϕ is, as follows:

- The upper disjunct quantifies over non-final states of S_r. We consider all ϵ paths that end in some state s', from which we transition to t with an A transition. We account for the tests encountered along our ϵ-path via conjunctive transitions into the relevant automata.
- The lower disjunct quantifies over ϵ paths that end in f_r, the accepting state of our marked ϵ-NFA. Consider that the A must still be processed in the automaton for ψ, so we redirect edges going in to f_r to the successors of the initial state of the automaton for ψ (with an A-transition). As before, we account for the tests with suitable conjunctive transitions.

The definition of the automaton for $[\phi]\psi$ is dual. Suppose ϕ contains k tests $\theta_1, \dots, \theta_k$. We construct the automata for the negations of these tests – let these be $\sigma_1, \dots, \sigma_k$. Then,

$$A_{[\phi]\psi} = (\Sigma, S_r \cup S_\psi \cup S_1 \cup \ldots \cup S_k, \{s_r^0\}, \rho, S_r \cup F_\psi \cup F_1 \cup \ldots \cup F_k)$$

where the S_i, F_i refer to the states and initial states of σ_i (for $1 \leq i \leq k$). Notice that every state in S_r is now accepting, since we do not want to reject executions that never match the regular expression at all. Our modified transition relation is now

$$\rho(s, A) = \begin{cases} \rho_\psi(s, A) & s \in S_\psi \\ \rho_j(s, A) & s \in S_j, j \in \{1, \dots, k\} \\ \bigwedge_{s' \in S_r \setminus \{f_r\}} \bigwedge_{\pi \in \Pi(s,s')} \bigwedge_{t \in \rho_r(s', A)} \left(t \vee \bigvee_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right) & \\ \wedge & s \in S_r \\ \bigwedge_{\pi \in \Pi(s, f_r)} \left(\rho_\psi(q_\psi^0, A) \vee \bigvee_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right) & \end{cases}$$

In terms of model checking, we take $\Sigma = 2^{AP}$, where AP is a set of atomic propositions that are relevant to the system under analysis. The alphabet thus consists of *subsets* of the atomic propositions (in practice, the atomic propositions that are true at each step in the path being considered).

We illustrate this with an example – let us construct the automaton for $\langle (p? + q?) \rangle r$. We first construct the sub-automata for the tests, the suffix as well as the regular expression. Constructing a marked ϵ-NFA for the tests $p?$ and $q?$ is simple, as shown in (1) in Figure 2.10. We can then compose these marked ϵ-NFAs using Thompson's construction (discussed in Section 2.6.1), yielding (2) in Figure 2.10. We can also construct the automata for the tests and the suffix, which is (3) in Figure 2.10. Now, we can build the alternating automaton for $\langle (p? + q?) \rangle r$.

Consider s_1. Observe that the automaton (2) itself does not have any non-ϵ transitions, so regardless of which atomic propositions are true in the current state, the first disjunct of equation 2.1 does not apply. We then consider the second disjunct; $f_r = s_4$, and we observe that there are precisely 2 ϵ-paths from s_1 to s_4 – $\pi_1 = s_1, s_2, s_4$ and $\pi_2 = s_1, s_3, s_4$. Now notice that $m(\pi_1) = \{p\}$ and $m(\pi_2) = \{q\}$. Also, defining $\theta_1 = p$, $\theta_2 = q$, observe that we have

$$\rho_\psi(s_\psi^0, A) = \begin{cases} \top & r \in A \\ \bot & r \notin A \end{cases}$$

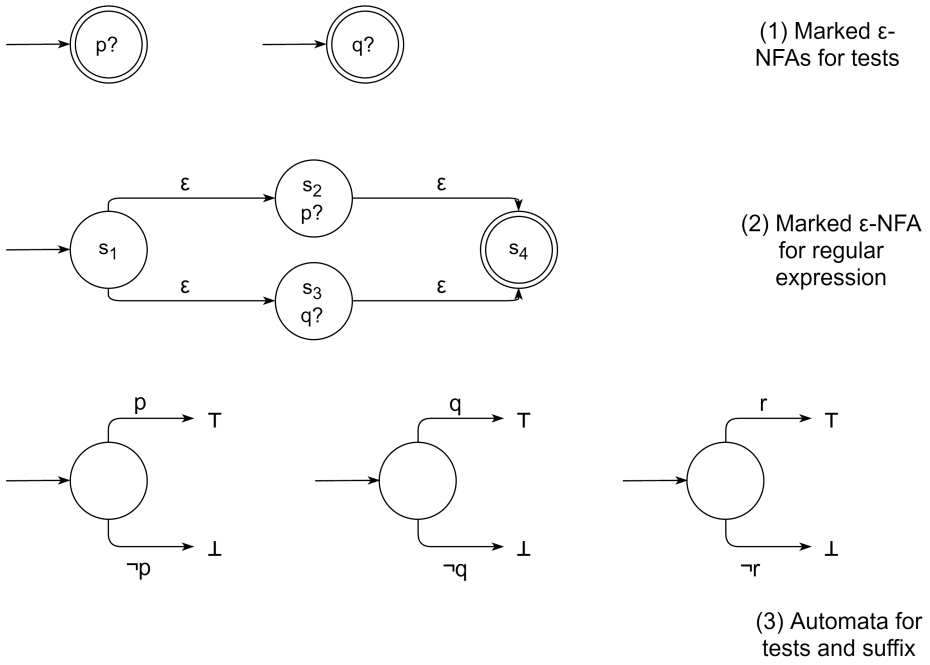


Figure 2.10: Sub-automata for constructing an alternating automaton for $\langle (p? + q?) \rangle r$.

$$\rho_1(s_1^0, A) = \begin{cases} \top & p \in A \\ \bot & p \notin A \end{cases}$$

$$\rho_2(s_2^0, A) = \begin{cases} \top & q \in A \\ \bot & q \notin A \end{cases}$$

And hence

$$\begin{aligned} \rho(s_1, A) &= (\rho_\psi(s_\psi^0, A) \wedge \rho_1(s_1^0, A)) \vee (\rho_\psi(s_\psi^0, A \wedge \rho_2(s_2^0, A)) \\ &= \rho_\psi(s_\psi^0, A) \wedge (\rho_1(s_1^0, A) \vee \rho_2(s_2^0, A)) \qquad \text{by distributivity} \end{aligned}$$

This captures the intended meaning (that we need r, and either p or q true in the initial state of this path of our system's evolution). Supposing the current symbol in our alphabet (corresponding to a state of the system being analysed) satisfies p and r, but not q. Then, we can construct an accepting run of $A_{\langle p?+q? \rangle r}$, as the $\top$ transition is available – we can reach the $\top$ transition in ρ_ϕ and ρ_1. Conversely, if we only had r satisfied, we have to transition to $\bot$ regardless of which of the test automata we choose to transition into – and thus there cannot be an accepting run for any path starting in such a state.

(Note that we can similarly define the transition relation ρ over the rest of the states, such as s_2 through s_4 from the automaton, and they are a part of the model. They are not required for our analysis here, though.)

2.6.4 Breakpoint Construction

Alternating automata are useful in that they provide a relatively succinct representation of a property that is to be verified, as compared to nondeterministic Büchi automata [93]. However,

directly checking these automata for the existence of run trees appears difficult; one way to deal with this complexity involves converting them to equivalent nondeterministic Büchi automata [21].

A possible approach for this conversion is known as the *breakpoint construction* and was first outlined in [73]; we use a simpler version of the formulation in [20]. Let $A = (2^{AP}, S, s^0, \rho, F)$ be an alternating Büchi automaton; there exists a nondeterministic Büchi automaton $A' = (2^{AP}, S', s^{0'}, \rho', F')$ that accepts a path if and only if A also accepts it (i.e. $L_\omega(A) = L_\omega(A')$). Furthermore, we define the set of rejecting states $S_N = S \backslash F$. Then, A' can be constructed as follows:

- $S' = 2^S \times 2^{S_N}$
- $s^{0'} = (s^0, \emptyset)$
- $F' = \{2^S \times \{\emptyset\}\}$
- Furthermore, we define ρ' such that $(L', R') \in \rho((L, R), A)$ where $A \in \Sigma$, where $L = \{l_1, \ldots, l_n\}$ if there exist $L'_1, \ldots, L'_n$ satisfying the following:
  - L'_j satisfies $\rho(l_j, A)$ for $j \in \{1, \ldots, n\}$
  - $\bigcup_j L'_j = L'$
  - $R' \subseteq L'$
  - either $R = \emptyset$ and $R' = L' \cap S_N$, or $R \neq \emptyset$ and $R' \cup (L' \cap F)$ satisfies $\bigwedge_{r \in R} \rho(r, A)$

Intuitively, the states of A' consist of pairs of subsets of the states of A – each pair has a Left set L and Right set R. The sets $L'_1, \ldots, L'_n$ used to construct the Left set L encode the expansions of a run tree of the original alternating automaton A; the Right set R consists of rejecting states visited since the last time R was empty. For example, if for every state $r \in R$ we have that $\rho(r, A)$ is satisfiable by L', and every state in L' is accepting, then we have $L' \cup F = L'$, and so we can choose $R' = \emptyset$.

We walk through a portion of the breakpoint construction for the simple alternating Büchi automaton introduced in Section 2.6.2. The transition relation of that automaton was

$$\rho(s_i, a) = \begin{cases} s_1 \wedge s_2 & i = 1, 3 \\ s_1 \vee s_3 & i = 2 \end{cases}; \rho(s_i, b) = s_i \text{ for all } i. \tag{2.2}$$

Consider Figure 2.11:

- For the starting state $(\{s_1\}, \emptyset)$, if the current symbol is a, we need L' to satisfy $\rho(s_1, a) = (s_1 \wedge s_2)$. Suppose we pick $L' = \{s_1, s_2\}$. Then, because $R = \emptyset$, we have R' as the rejecting states in L' i.e. $R' = \{s_2\}$. If the current symbol is b, then any state with $s_1 \in L'$ is acceptable.
- For the state $(\{s_1, s_2\}, \{s_2\})$, if the current symbol is a we need the new L' to have a subset which satisfies $\rho(s_1, a)$ and a subset which satisfies $\rho(s_2, a)$. We can pick $L' = \{s_1, s_2\}$ again. However, this time because $R \neq \emptyset$, we need $R' \cup \{s_1\}$ to satisfy $\rho(s_2, a)$. We can choose $R' = \emptyset$; hence, we can transition to $(\{s_1, s_2\}, \emptyset)$. If the current symbol is b, we can pick $L' = \{s_1, s_2\}$ again, but we need $R' \cup \{s_1\}$ to satisfy $\rho(s_2, b) = s_2$, so we again need $R' = \{s_2\}$.

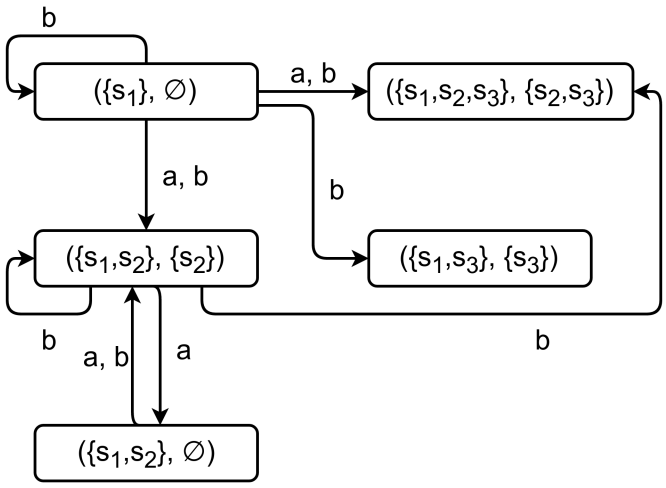


Figure 2.11: Portion of the nondeterministic Büchi automaton generated by the breakpoint construction on the automaton with transition relation in Equation 2.2. We do not include the entire nondeterminstic Büchi automaton owing to its size.

- For the state $(\{s_1, s_2\}, \emptyset)$, the requirements on L' are similar to $(\{s_1, s_2\}, \{s_2\})$. We need the new L' to contain both s_1 and s_2. If we pick $L' = \{s_1, s_2\}$, then since $R = \emptyset$, we have $R' = \{s_2\}$ in either case.

This breakpoint construction may be carried out explicitly, though as the above construction has an exponential blowup in the number of states (possibly $O(3^n)$ in the worst case [20], since each automaton state may either be in L, in both L and R or neither) we prefer to use the symbolic encoding introduced in [20].

2.7 Existing Model Checkers

In this section we briefly outline several existing model-checking tools. We consider their capabilities for handling various specification languages and, where appropriate, the algorithms they use.

2.7.1 MCMAS

MCMAS is an open source model checker for multi-agent systems developed at Imperial College London; it implements symbolic model checking with binary decision diagrams using the CUDD package [65, 10]. MCMAS accepts input in the form of Interpreted Systems Programming Language (ISPL) specifications, and is able to verify CTLK and ATLK properties. Extensions exist that allow verification of properties with unbounded numbers of agents, verification of properties in strategy logic, or use sentential decision diagrams (SDDs) to boost performance [10].

We discuss MCMAS' existing architecture in greater detail in Section 2.8; this is important as we will be implementing our tool as an extension to MCMAS.

2.7.2 MCK

MCK was the first symbolic model checker supporting temporal-epistemic specifications. It supports other approaches as well, such as explicit state and bounded model checking [5].

The model checker supports CTL*K specifications; in addition to the observational semantics supported by MCMAS, it also supports clock semantics and perfect recall semantics for a fragment of these specification languages [53]. Tests have suggested that MCK may be less efficient than MCMAS on large state-spaces, however [65].

2.7.3 NuSMV

NuSMV is one of the most commonly used symbolic model checkers [65], and one of the top-performing model checkers in terms of speed and accuracy [78]. It supports BDD-based symbolic model checking as well as SAT-based and bounded model checking for CTL and LTL specifications [8]. NuSMV does not readily support epistemic specifications or interpreted systems 'out of the box', though it is possible to translate interpreted systems into SMV code [76] which NuSMV can then verify. It is worth noting that the algorithm that NuSMV uses to verify LTL specifications involves the *tableau construction* presented in [28].

Furthermore, NuSMV also supports parsing of Property Specification Language (PSL) specifications, a temporal logic that also supports regular expressions. However, as of version 2.6, it only supports the verification of PSL specifications that can also be expressed in LTL or CTL [26].

2.7.4 VerICS

VerICS is a model checker for real-time specifications and multi-agent systems that supports SAT-based approaches [100]. Unlike MCMAS which attempts verification by analysing all possible states of a system, VerICS focuses on bounded model checking; the techniques are fundamentally different and are arguably complementary [65]. VerICS does support CTL* specifications, as well as partial support for LTLK as defined in this report[3] and CTL extended with a variety of epistemic logics.

2.7.5 SPIN

SPIN is a tool allowing for verification of *asynchronous process systems*; it can accept a system specification in Process Meta Language (PROMELA) and verifies properties specified in LTL. It builds a Büchi automaton for the system as well as for the property (as described in 2.5.1) and checks for non-emptiness of their product [50]. Notice that agents in interpreted systems tend to evolve synchronously while SPIN focuses on asynchronous systems. There also does not appear to be native support for epistemic specifications.

2.8 Deeper Investigation of MCMAS

We decided to extend the model-checker MCMAS to support formulae in the additional specification languages. This allowed us to re-use various components of MCMAS, such as the lexer and parser, system by which agents and states were encoded, as well as the algorithms for computing reachable states. (Of course, we would need to modify or extend some parts, such as the lexer and parser, to accept expressions in the new specification languages.)

This section contains a more involved investigation into the usage and architecture of MCMAS (specifically, version 1.2.2 which was the latest version at the time we started the project).

[3] Based on [11] distributed knowledge does not seem to be supported, and common knowledge cannot be expressed succinctly.

Understanding the architectural as well as low-level implementation details of MCMAS was instrumental in being able to successfully extend its capabilities to handle the new specification languages while, as far as possible, maximising code reuse and avoiding reimplementing existing functionality.

2.8.1 Usage

MCMAS is distributed as an open source tool; it may be downloaded as a zipped file from [10]. The tool itself is packaged with user documentation which explains in greater detail how it should be used. In this section we thus consider and highlight the features most relevant to our investigation.

MCMAS is typically invoked from the command line; the user optionally specifies one or more flags that control its execution, as well as the path of the ISPL file to be checked. One can invoke MCMAS with the `-h` flag to list all of the valid options. For example, we could run MCMAS with the following command on `bit_transmission.ispl`, an ISPL file representing the bit transmission protocol as in Appendix A (though with fewer formulae), where we request MCMAS to also output counterexamples (`-c 1`).

```
$ ./mcmas -c 1 bit_transmission.ispl
```

MCMAS then automatically determines whether each of the formulae holds as well as suitable witnesses or counterexamples; more detail concerning *how* it does so may be found in Section 2.8.2. It prints suitable output:

```
1  ************************************************************************
2                          MCMAS v1.2.2
3
4   This software comes with ABSOLUTELY NO WARRANTY, to the extent
5   permitted by applicable law.
6
7   Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
8   Please send any feedback to <mcmas@imperial.ac.uk>
9  ************************************************************************
10
11 Command line: ./mcmas -c 1 bit_transmission.ispl
12
13 bit_transmission.ispl has been parsed successfully.
14 Global syntax checking...
15 Done
16 Encoding BDD parameters...
17 Building partial transition relation...
18 Building BDD for initial states...
19 Building reachable state space...
20 Checking formulae...
21 Building set of fair states...
22 Verifying properties...
23   Formula number 1: (AG (bit0 || bit1)), is TRUE in the model
24     -- Sorry it is not possible to compute witnesses for non-ECTLK formulae
25   Formula number 2: (EF recbit), is TRUE in the model
26   The following is a witness for the formula:
27    < 0 1 >
28   States description:
29 ------------- State: 0 -----------------
30 Agent Environment
31   state = none
32 Agent Sender
```

```
33    ack = false
34    bit = b0
35  Agent Receiver
36    state = empty
37  ----------------------------------------
38  ------------- State: 1 -----------------
39  Agent Environment
40    state = SR
41  Agent Sender
42    ack = false
43    bit = b0
44  Agent Receiver
45    state = r0
46  ----------------------------------------
47  done, 2 formulae successfully read and checked
48  execution time = 0.01
49  number of reachable states = 18
50  BDD memory in use = 9018048
```

Observe that MCMAS indicates that both formulas hold in the model (lines 23, 25), as well as a witness path for $EFrecbit$ (lines 26 through 46). Since $AG(bit0 \vee bit1)$ holds in the model, it holds over every possible execution, and MCMAS thus does not report a witness. MCMAS also prints several metrics concerning runtime and memory usage (lines 48 through 50); we will be analysing these, as well as several additional metrics (which will be printed if we set the `-v` (verbosity) flag to at least a certain level) when performing our experimental evaluation in Chapter 8.

2.8.2 Architecture

At a high level, MCMAS accepts an ISPL file as input and determines if the temporal logic specifications (normally provided in CTLK or ATLK) in the ISPL file hold. Figure 2.12 highlights the intermediate steps taken by MCMAS when verifying such specifications, which are as follows:

1. MCMAS reads an ISPL file as input, and parses it using `lex` and `yacc` to determine the structure of the ISPL file. In addition to this, various command-line arguments can be provided, which can control several parameters of MCMAS' execution such as the heuristics used to decide on variable ordering for BDDs, or the level of verbosity with which output should be printed. Greater detail about the syntax of ISPL may be found in Definition 2.45.

2. MCMAS then encodes the interpreted system $\mathcal{I}$ using BDDs. This is done by iterating through the agents and allocating BDD variables to them (as well as the environment). For each agent, we allocate two separate sets of variables to reflect the agent's local states in the current and next states (following the symbolic model checking procedure described in Section 2.4.3). We also allocate variables representing the actions a given agent might select on each evolution, given the protocol specified for that agent. This is used to build a transition relation R for our interpreted system $\mathcal{I}$.

3. MCMAS then computes the set of states reachable in the model; most of the logic is found in `computereach.cc`. This is done by (symbolically) computing the least fixed point of $f(X) = InitStates \cup succ(X)$ where $succ(X)$ refers to the successors of X. It is worth noting that this step can be fairly time-consuming; it typically requires the most time out of any of the steps in the model-checking process.

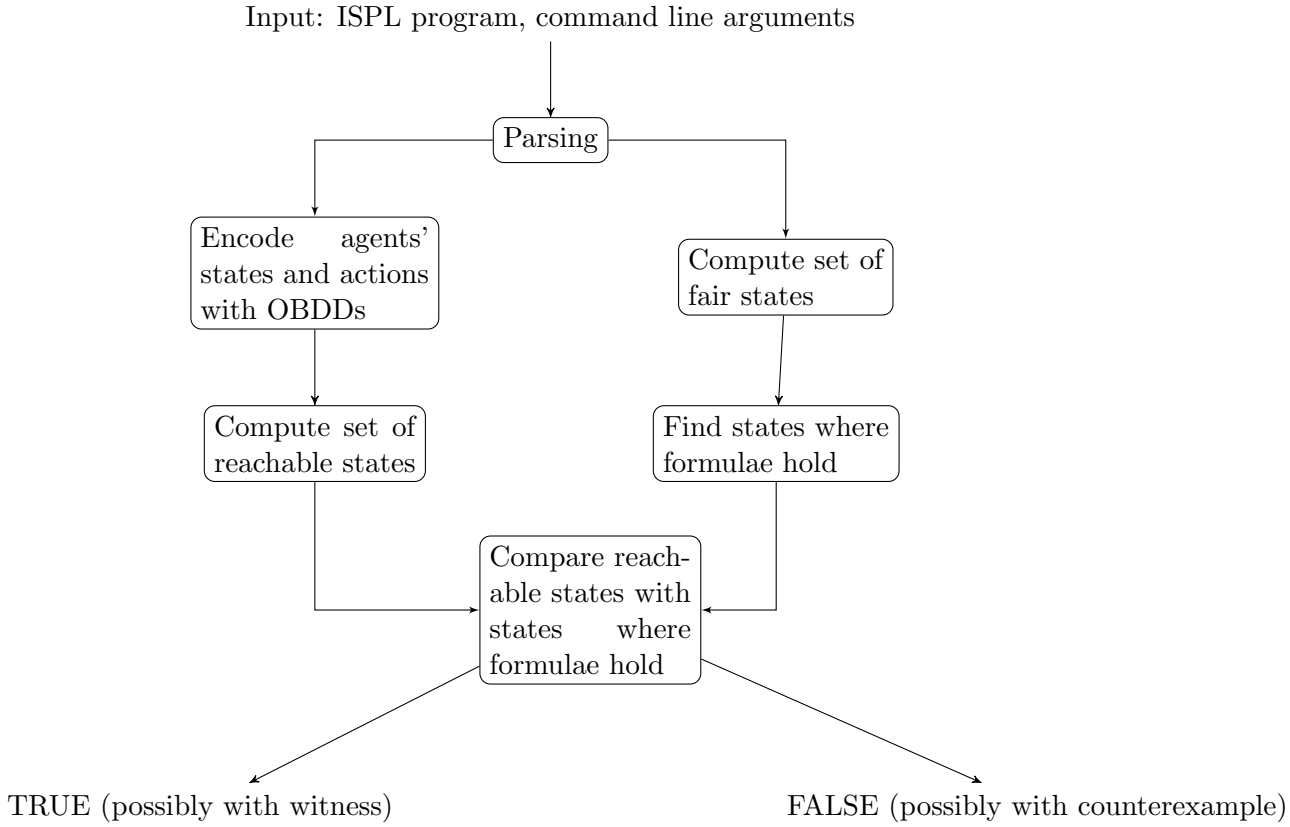


Figure 2.12: High level architecture of MCMAS.

4. If fairness constraints are present in the model, MCMAS calculates the states that are the start of a fair path (in `model_checking.cc`). This is done by building BDDs for each of the fairness properties in the ISPL program, representing the states in which they hold, and then performing a fix-point calculation finding the states in which there exist paths where each fairness property holds infinitely often (which is detailed in [29]).

5. MCMAS then finds the states in which the temporal-epistemic logical formulae in the ISPL program hold. The `modal_formula` class is used to represent a modal formula, and calling `check_formula()` on a given instance returns a BDD corresponding to the states in which the relevant formula holds. For CTLK operators, MCMAS implements an optimised variant of the algorithm discussed in Section 2.3.2 (in particular, it implements the optimisation with EG discussed in [66]) extended to handle epistemic operators. MCMAS also supports ATLK specifications, the details of which are not the focus of this project.

6. Finally, MCMAS compares the reachable states with the states in which the logical formulae hold (in other words, it compares the results of steps 3 and 5). This is carried out by, for a specification formula ϕ, constructing the formula $V_\phi = InitStates \rightarrow \phi$ and (symbolically) computing the reachable states in which V_ϕ holds, comparing it against the overall set of reachable states. The specification is decided to be TRUE in the model if and only if the two are equivalent; otherwise, the specification is decided as FALSE. This is correct, since if the two are not equivalent, it must be the case that there exists a reachable state g in which V_ϕ does not hold (since the set of reachable states in which V_ϕ holds is by definition a subset of the set of reachable states). Then, we have $\mathcal{I}, g \nvDash InitStates \rightarrow \phi$, which holds if and only if g is both an initial state and $\mathcal{I}, g \nvDash \phi$; hence, there is an initial state in which ϕ does not hold, meaning the specification ϕ is false in the model.

7. Optionally, the user may generate witnesses and/or counterexamples, by specifying a command-line flag `-c`. This may be useful in understanding *why* the model satisfies or does not satisfy a specification. If this flag is specified, then

   - If a formula using only non-negated existential quantifiers (an ECTLK formula) is true (e.g. $EG\phi$), then MCMAS generates and outputs a witness (an execution on which the formula is satisfied).
   - If a formula using only non-negated universal quantifiers (an ACTLK formula) is false (e.g. $AF\phi$), then MCMAS generates and outputs a counterexample (an execution on which the formula is not satisfied).

MCMAS does also have several other capabilities, such as allowing the user to manually explore the system with a simulation, or to try different algorithms for selecting BDD variable orderings. Some of these alternative options are displayed if one runs MCMAS with the `-h` flag; these additional capabilities are also documented in [7].

2.8.3 Interpreted Systems Programming Language (ISPL)

MCMAS accepts specifications in Interpreted Systems Programming Language (ISPL). The general structure of an ISPL file is as follows, from [7]:

Definition 2.45. (ISPL Syntax) The syntax of an Interpreted Systems Programming Language (ISPL) file at a high level is as follows:

```
1 Semantics = MultiAssignment | SingleAssignment
2 Agent Environment
3   Obsvars:
```

```
4       ...
5     end Obsvars
6     Vars:
7       ...
8     end Vars
9     RedStates:
10      ...
11    end RedStates
12    Actions = {...};
13    Protocol:
14      ...
15    end Protocol
16    Evolution:
17      ...
18    end Evolution
19  end Agent
20
21  Agent TestAgent
22    Lobsvars = {...};
23    Vars:
24      ...
25    end Vars
26    RedStates:
27      ...
28    end RedStates
29    Actions = {...};
30    Protocol:
31      ...
32    end Protocol
33    Evolution:
34      ...
35    end Evolution
36  end Agent
37
38  Evaluation
39    ...
40  end Evaluation
41
42  InitStates
43    ...
44  end InitStates
45
46  Groups
47    ...
48  end Groups
49
50  Fairness
51    ...
52  end Fairness
53
54  Formulae
55    ...
56  end Formulae
```

We focus particularly closely on the sections of the syntax above that are pertinent to our extension of MCMAS to support model checking of additional specification languages.

- The `Agent` declarations allow users to specify the environment as well as relevant agents (note that `TestAgent` is simply an agent name, and is not a reserved word). In particular, `Obsvars` in the environment refer to variables observable by all agents; an agent can also

specify `Lobsvars` referring to environment variables it can see. `RedStates` is used in deontic properties (reflecting the states which are considered incorrect behaviour for the underlying agent) and is not relevant to our work.

- The `Evaluation` section (lines 38–40) allows users to specify boolean variables to be used when specifying fairness specifications as well as formulae. These variables typically involve Boolean combinations of relational expressions over variables in the program and help to make specifications more succinct. For example, using the Bit Transmission Protocol example discussed in Section 2.2.2, we can define an atomic proposition `recack if (Sender.state = 0A or Sender.state = 1A)`. This is significant as the atomic propositions used in specifications are not directly represented as BDD variables; we will need to adapt the symbolic algorithm presented in [28] when computing the product of the tableau and the interpreted system under investigation.

- The `InitStates` section (lines 42–44) allows the users to specify precisely which states are valid starting states in the system. We are not concerned with whether the formulae to be verified hold in states that are not reachable from any of the starting states.

- The `Groups` section (lines 46–48) allows users to specify groups of agents, which will be referenced when specifying properties using the grouped epistemic modalities such as E_Γ, D_Γ and C_Γ.

- The `Fairness` section (lines 50–52) allows users to specify fairness conditions, which are Boolean combinations of relational expressions over program variables. These expressions must hold infinitely often along all execution paths of the system, and are to be respected when model checking the additional specification languages as well. Note that this is particularly important for CTLK because the specification language by itself cannot express properties holding on paths satisfying fairness constraints.

- The `Formulae` section (lines 54–56) contains the list of formulae the system will be checked on. In the base version of MCMAS, CTLK and ATLK formulae with deontic operators are permitted.

Chapter 3

Linear Temporal Epistemic Logic (LTLK)

In this chapter we introduce a model checking technique for Linear Temporal Logic with epistemic modalities (LTLK), which was introduced in Section 2.3.5. We discuss the recursive algorithm of [70], and show that the adaptation preserves the time complexity of LTL model checking, which is exponential in the size of the formula, but linear in the size of the model.

We then discuss details of our symbolic implementation of LTLK model checking over interpreted systems, including counterexample generation. We implemented this as an extension of the MCMAS model checker. We conclude with a brief comparison of our tool with MCK; although verifying LTLK is already supported by MCK, how it is implemented in MCK is unclear.

The original work discussed in this chapter that was developed as part of this individual project is as follows:

1. **Extension of MCMAS to support LTL semantics.** MCMAS previously only had support for CTL and ATL, which are branching-time logics; properties about paths (e.g. "if p happens at some point, then q also does") would previously not have been verifiable.

2. **Integration and extension of the abilities of MCMAS to handle epistemic modalities with LTL semantics.** We further extended our LTL extension of MCMAS to support epistemic modalities, allowing us to support LTLK in full.

3. **MCMAS counterexample generation bugfix.** We fixed a bug concerning counterexample generation for $EG\top$ in MCMAS. This may have arisen in practice, even for CTLK properties, and is thus useful even outside of the context of our LTL implementation.

3.1 Algorithm

3.1.1 Recursive Descent over Epistemic Modalities

We solve the model-checking problem for LTLK by reducing it to the problem of model-checking plain LTL; we can then use the *tableau construction* method described in Section 2.5.2 to reduce *that* to the CTL model-checking problem with fairness constraints, which is already supported by MCMAS. This reduction can be carried out by adapting the recursive descent-based ideas of [70], which worked on verifying LTLK with *bounded model checking*, in the context of symbolic model checking with OBDDs.

For a given LTLK formula ϕ and interpreted system $\mathcal{I}$, we seek to determine the states g in which $\mathcal{I}, g \vDash \phi$ holds; that is, $[\phi]_{\mathcal{I}}$. We first compute the set of states in which each top-level epistemic subformula $K_i\psi, E_\Gamma\psi, D_\Gamma\psi$ or $C_\Gamma\psi$ of ϕ holds. We then treat these as atomic propositions when computing the states in which ϕ holds. This can be done recursively; since formulas are finite, they must have a finite nesting depth, and hence the recursion is finitely deep and we will eventually reach a formula that is a plain LTL formula, which we know how to check. For example, consider the LTLK formula

$$\phi = G\left(K_a(F(E_{\{a,b\}}(Gp)) \to Gq)\right) \tag{3.1}$$

where p, q are atomic propositions. We can compute the set of states in which ϕ holds, as follows.

1. First compute the set of states in which Gp (in red) holds using the tableau method (this is an LTL formula).
2. Then compute the set of states in which $E_{\{a,b\}}(Gp)$ (in blue) holds; we know the states in which Gp holds and can thus compute this using the epistemic accessibility relation for the agents a and b.
3. Then compute the set of states in which $F(E_{\{a,b\}}(Gp)) \to Gq$ (in green) holds. This is equivalent to finding the set of states in which $Fr \to Gq$ holds, where r is a fresh atomic proposition holding in precisely the states where $E_{\{a,b\}}(Gp)$ held. This is an LTL formula and we can thus use the tableau method.
4. Then compute the set of states in which $K_a(F(E_{\{a,b\}}(Gp)) \to Gq)$ (in purple) holds. This can be computed using the epistemic accessibility relation, as in step 2.
5. Finally compute the set of states in which ϕ holds; as in step 3, this is equivalent to finding the set of states in which Gr' holds, where r' is another fresh atomic proposition holding in precisely the states where $K_a(F(E_{\{a,b\}}(Gp)) \to Gq)$ held. Again, this is an LTL formula and we can thus use the tableau method.

(In practice, this is implemented in a top-down rather than a bottom-up manner, though in the interest of clarity of explanation we opted with a bottom-up presentation here.)

3.1.2 Complexity Analysis

It is known that LTL model checking is linear in the size of the model but exponential in the size of the formula – it is $O(2^{|\phi|} \times |\mathcal{I}|)$ [84]. However, the algorithm in question requires Büchi automata to be computed on-the-fly, which does not work well with symbolic model checking. The tableau construction for $\neg\phi$ itself generates $O(2^{|\phi|})$ states, though we do not need to construct these states explicitly. Thereafter, we check $(EG(\top))$ over the model with additional fairness constraints – the "model" here, however, is the product of the original system as well as the tableau (Definition 2.35) and thus has $O(2^{|\phi|} \times |\mathcal{I}|)$ states. CTL model checking is linear in the size of the model, formula and number of fairness constraints [13, 84]. We have linearly many fairness constraints in the worst case (recall that we generate one fairness constraint for every subformula involving a F, G or U modality); thus this step requires time $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$. We then need to check whether any states returned are in $sat(\phi)$, as well as perform negation – these steps are certainly bounded by $O(2^{|\phi|})$ in the worst case (we can compute $sat(\phi)$ and then explicitly test membership). Hence our LTL algorithm runs in time $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$; this is suboptimal but we believe this is outweighed in practice by the empirical efficiency of symbolic

model checking, as suggested in [78].

We now show that the recursive descent approach presented above for LTLK preserves the $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$ time complexity. We first introduce a lemma:

Lemma 3.1. Suppose that for $i \in 0, \ldots, k$ we have $f(y_i, n) = O(n2^{y_i \log y_i})$ and $0 \leq y_i < x$. Further suppose that $f(x, n) = \left(\sum_{i=0}^{k} f(y_i, n)\right) + O(n2^{x \log x})$. Then, $f(x, n) = O(n2^{x \log x})$.

Proof. From the definition of big-O notation we have for each i, $f(y_i, n) \leq c_i n 2^{y_i \log y_i}$ for some positive c_i. Then, observe that

$$
\begin{aligned}
f(x,n) &= \left(\sum_{i=0}^{k} f(y_i, n)\right) + O(n2^{x \log x}) \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{y_i \log y_i}\right) + O(n2^{x \log x}) && \text{by above assertion} \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{y_i \log y_i}\right) + cn2^{x \log x} && \text{for some positive } c\text{, by definition} \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{x \log x}\right) + cn2^{x \log x} && \text{since for each } i,\ 0 \leq y_i < x \text{ and so } 2^{y_i \log y_i} \leq 2^{x \log x} \\
&= \left(\left(\sum_{i=0}^{k} c_i\right) + c\right) n 2^{x \log x}
\end{aligned}
$$

Furthermore, since every c_i and c are positive, their sum is positive. Thus $f(x, n) = O(n2^{x \log x})$.

Observe that $|\mathcal{I}|$ is constant, while $|\phi|$ is decreasing after each recursive call. Hence, we define $f(|\phi|, |\mathcal{I}|)$ as the time taken to evaluate a path formula ϕ with model size $|\mathcal{I}|$, which is constant throughout. Furthermore, observe that combining previous solutions involves two steps:

- Assign fresh atomic propositions. The number of propositions is clearly bounded linearly in the size of the formula, since each proposition requires an appropriate subformula. Using pointers to the existing sets of states, or other suitable data structures, this step as a whole runs in $O(|\phi|)$ time.
- We then either need to compute the set of states over which an epistemic modality holds, using the epistemic accessibility relation (e.g. step 2 or 4 in our example formula 3.1), or perform model checking of an LTL formula (steps 1, 3 and 5).
  - Computing the set of states over which an epistemic modality holds may be done by considering the equality of local states as far as an agent or group of agents is concerned [66]. Assuming that the epistemic accessibility relations are pre-computed, we can determine which states these hold in in $O(|\mathcal{I}|)$ time.
  - Performing model checking of an LTL formula with $O(|\phi|)$ fairness constraints requires $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$ time (as discussed at the beginning of this subsection).

  Thus, this step runs in $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$ time.

Thus, Lemma 3.1 applies and we have that the time complexity of our algorithm is $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$. Note that if the LTL algorithm ran in $O(2^{|\phi|} \times |\mathcal{I}|)$, then our recursive descent algorithm would also run in $O(2^{|\phi|} \times |\mathcal{I}|)$ time; the proof is similar, using Lemma C.1 instead.

3.2 Implementation

At a high level, we needed to modify the following aspects of MCMAS:

- Add support for parsing formulae specified in LTLK (Section 3.2.1).
- For each formula ϕ, construct the symbolic tableau for ϕ (Section 3.2.2).
- Compose the symbolic tableau with the Kripke structure for the model, which MCMAS has already computed (Section 3.2.3).
- Check $EG\top$ in the model, from states not in $sat(\phi)$; then negate the result (Section 3.2.4).
- Finally, add support for counterexample generation; if a formula ϕ is false, we want a counterexample trace (Section 3.2.5).

3.2.1 Expression Parsing

The only section of MCMAS's implementation of the ISPL grammar we needed to change would be the `Formulae` section; we would need to add the ability for users to specify LTLK formulae here. We adopted the relatively simple approach of requiring users to specify the specification language they were using, by prefixing their LTLK formulae with the new keyword `LTL`; a similar approach is used in other popular model checkers such as NuSMV which requires users to specify `LTLSPEC` (instead of simply `SPEC` for CTL specifications) [8]. This *is* otherwise a valid identifier (albeit unlikely), so it is possible that an extremely small subset of old models will no longer work if this change is made to the grammar.

In terms of implementation, MCMAS uses `lex` and `yacc` to split an input ISPL file into tokens and parses these tokens. The following changes were made to the grammar:

1. As discussed in the first paragraph, a new `LTLPREFIX` terminal was added, which is "`LTL`".
2. The `formlist` nonterminal, used to parse line 55 in the sample syntax listing in Definition 2.45, was extended with rules allowing sequences with an `LTLPREFIX` terminal followed by a `ltlformula` nonterminal, and parsing an LTLK formula from the nonterminal.
3. The `ltlformula` nonterminal has several production rules corresponding to the syntax of LTLK formulae as defined in Definition 2.22. For usability reasons, we support common propositional abbreviations (such as $\vee$ or $\rightarrow$) as well as the G and F temporal connectives.
4. Both `ltlformula` and the original `formula` nonterminal (which supported CTLK and ATLK specifications) require a way of specifying atomic propositions. As the parsing logic for this is nontrivial (94 lines of code), we refactored these rules to a new `terminalformula` nonterminal which is a possible outcome of parsing an `ltlformula` or `formula`.
5. We decided to duplicate the definitions of `epistemicprefix` and `gepistemicprefix`, the nonterminals internally used when parsing an individual or group epistemic modality such as `K(a, B)` where `a` is an agent's name and `B` is a formula. I thus made separate `epistemicprefixltl` and `gepistemicprefixltl` nonterminals.

   This was to avoid requiring users to specify that they were using LTLK repeatedly; otherwise, specifications such as `LTL G K(a, F p)` would result in syntax errors (as the grammar expects a standard `formula`, not an `ltlformula` in the bracket) – the alternative of

requiring users to write `LTL G K(a, LTL F p)`, as well as permitting mixed specifications (e.g. `LTL G K(a, AG EF p)`) was deemed unacceptable. Note that the algorithm is perfectly able to cope with mixing LTL and CTL specifications across epistemic modalities; this decision was made in the interest of usability, not because of any limitations of the algorithm itself.

An interesting feature of our parsing is that when interpreting a `ltlformula`, `epistemicprefixltl` or `gepistemicprefixltl`, we treat this nonterminal as a node in the formation tree of our formula. For example, code for one of the production rules is as follows:

```
| LTLPREFIX ltlformula SEMICOLON {
  if($2!=NULL) {
    is_formulae->push_back(modal_formula(60, $2));
  }
}
```

This constructs a `modal_formula` with operator 60 (this is a `char` value used to distinguish between operators internally in MCMAS) and first argument being the result of parsing the `ltlformula` nonterminal. This allows us to know more easily when we need to use the tableau construction, and also yields a subsequent easier transition to model checking CTL*K formulae; this operator may easily be re-used as the CTL*K path quantifier A.

MCMAS also carries out a semantic check after formulae are parsed in `syntaxcheck.cc`. The `modal_formula` class representing modal formulae has a `check_atomic_proposition()` method, which verifies that all atomic propositions appearing in each formula are defined. I extended this method to handle LTLK expressions appropriately.

3.2.2 Tableau Construction

We implemented LTLK model checking with the tableau construction detailed in Section 2.5.2. This involved creating a new utility class `ltlk.cc`, which included several utility methods. These methods recursively compute several components of the tableau construction. The key new methods added are as follows:

```
modal_formula* remove_path_gs(modal_formula* formula);
set<modal_formula*>* get_elementary_formulas(modal_formula* formula);
BDD check_path_sat(modal_formula* formula, bdd_parameters* para,
                   const map<modal_formula*, BDD>& index_map,
                   const map<modal_formula*, BDD>& next_index_map,
                   bool next);
BDD build_path_tableau(bdd_parameters* para,
                       const map<modal_formula*, BDD>& index_map,
                       const map<modal_formula*, BDD>& next_index_map);
```

1. `remove_path_gs()` is a preliminary step that rewrites expressions of the form $G\phi$ to $\neg F \neg \phi$, recursively traversing the formation tree of the specification. This made implementing some of the other methods easier; it would have been possible to rewrite $F\phi$ as $(\top U \phi)$ (where $\top$ would be represented with the BDD 1) but adapting the other methods to deal with F was considerably easier than doing so with G.

2. `get_elementary_formulas()` returns a set of pointers to elementary formulae (Definition 2.32) of a given modal formula. This function allocates new memory for formulae not part of the existing specification (in particular, when a $(\phi U \psi)$ specification is encountered, generating an elementary formula $X(\phi U \psi)$).

3. `check_path_sat()` implements the *sat* function in Definition 2.33, directly handling modalities of the form $F\phi$ as well. When invoked on a formula, it returns a BDD corresponding to the states in which it holds (provided it is an LTLK formula). The boolean argument indicates whether we are trying to compute the *sat* function over the variables referring to the current or next set of states.

4. `build_path_tableau()` constructs the tableau symbolically by building its transition relation (Definition 2.34). This method returns the transition relation as a BDD.

The map arguments in the latter two methods serve as a map of elementary formulas of the tableau to BDD variables in the new symbolic structure created.

3.2.3 Structural Composition

MCMAS checks the states in which a modal formula ϕ holds by calling its `check_formula()` method, which returns a BDD corresponding to the states in which said formula holds. To determine whether a specification holds in general, it checks if all of the initial states are part of the BDD returned; in practice this works by checking $InitStates \rightarrow \phi$ and checking that the resultant BDD is the BDD for 1.

Since this method is invoked recursively on input formulae, the use of a special operator indicating the root of an LTLK formula's parse tree (as discussed in Section 3.2.1) allows us to execute the same construction steps regardless of what the principal connective of the LTLK formula might be. At a high level, when `check_formula` is invoked on said operator, the method proceeds as follows:

1. `remove_ltl_gs()` as discussed in Section 3.2.2 and then `push_negations()` are used to remove G operators and simplify the formula, removing redundant negations.

2. A helper function `compute_epistemic_subformulae()` is used which carries out recursive descent of the tree as described in Section 3.1, computing a map between pointers to modal formulas and BDDs representing the states in which they hold. Subsequently, top-level epistemic subformulae are treated as elementary formulas which are true precisely in the set of states represented by the corresponding BDD.

3. `get_elementary_formulas()` is called on the output of `push_negations()`, extracting the elementary formulas. Suppose there are n such formulas.

4. $2n$ new BDD variables are declared using CUDD's BDD manager; we iterate through the elementary formulas and allocate 2 variables to each formula, one representing the elementary formula's current state and one representing its next state. We also need to supplement the `bdd_parameters` struct's `v` and `pv` vectors, referring to variables concerning the current and next state respectively, with the new variables[1]. Furthermore, during this step we build up two maps linking said elementary formulas to their BDD variables.

5. `build_ltl_tableau()` is called on the output of `push_negations()`, which constructs the transition relation for the tableau. This uses the maps populated in the previous step.

At this point, we need to compose the tableau and ISPL model, with the goal of finding suitable paths in both the model and tableau. However, the symbolic algorithm presented in [28] achieves consistency between model and tableau BDD variables by using the same BDD variable for both

[1]Many utility methods, such as taking preimages, make assumptions about the `bdd_parameters` struct.

the model and tableau in the case of atomic propositions. This is not directly possible here, because of the way MCMAS allocates BDD variables; in particular, atomic propositions in the tableau are Boolean variables specified in the `Evaluation` section. To enforce consistency in the transition relation, we augment the tableau with a set of *consistency rules*:

Definition 3.1. (Consistency Rules) Suppose p is a tableau variable that is true only in the states satisfying A. Then, a *consistency rule* for p is

$$T, g \vDash p \leftrightarrow \mathcal{I}, g \vDash A$$

This is implemented using a `build_consistency_rules()` method, which iterates through the elementary formulas and builds up a suitable BDD enforcing if-and-only-if constraints between the relevant tableau variable and its underlying meaning in the BDD of the model. The tableau then has these constraints added (implemented as `tableau *= rules`). The composition involves adding the tableau's transition relation to `para->vRT`, a vector of the various transition relations used by the agents that is used to compute the global transition relation. Notice that with the existing MCMAS architecture it is not necessary to enforce constraints between variables corresponding to the next states of the agents as well as next tableau variables, because the method for computing existential preimages (`exists_EX` in `utilities.cc`) already checks said correspondence in the next state before returning the existential preimage (though in general this check does need to be made at some point before returning meaningful results).

3.2.4 Path Finding in the Composed Model

We then construct the fairness constraints we need to apply to ensure that eventuality properties are satisfied; this is implemented through the use of a helper `get_ltl_fairness_constraints()` method which walks the formation tree of the formula and upon reaching the F or U connectives, adds a suitable fairness constraint (a BDD for the states that need to be visited infinitely often).

Next, we check the CTL specification $EG\top$ under the existing fairness constraints as well as the additional ones we generated. It is worth noting that some small steps are needed to keep the `bdd_parameters` struct consistent with the new model being examined, such that the utility methods for checking EG with fairness constraints work properly.

Next, recall that the tableau construction has the property that if a state satisfies $EG\top$ under the relevant fairness constraints, as well as $sat(\phi)$, then there exists an infinite path on which ϕ holds in the starting state. Thus, to check that all paths from a starting state satisfy ϕ, we need to check that *no path satisfies* $\neg\phi$. We thus compute a BDD for $sat(\neg\phi)$, compose it with the consistency rules and then find the states satisfying both $EG\top$ and $sat(\neg\phi)$. We compute the projection of this BDD to get the set of states *in the original model* from which there exists an infinite path with $\neg\phi$ holding in the starting state, and then take its complement to find the states from which no path has $\neg\phi$ holding in the starting state i.e. all paths have ϕ holding in the starting state. This may be best illustrated with the following code:

```
1 // infinite_paths is the BDD with EG T + fairness
2 BDD sat_neg_f =
3   !(new_formula->check_ltl_sat(para, index_map, next_index_map, false)) * rules;
4 infinite_paths *= sat_neg_f;
5
6 // We don't care about the starting states of the tableau
7 for (map<modal_formula*, BDD>::iterator it = index_map.begin();
8      it != index_map.end(); ++it) {
9   infinite_paths = infinite_paths.ExistAbstract(it->second);
```

```
10 }
11 for (map<modal_formula*, BDD>::iterator it = next_index_map.begin();
12      it != next_index_map.end(); ++it) {
13   infinite_paths = infinite_paths.ExistAbstract(it->second);
14 }
15 // We want the states satisfying A f = !E! f
16 result = !infinite_paths;
17 // (the product of result with the reachable states is returned)
```

We conclude with a small amount of clean-up work, resetting the global variable `agents` as well as BDD parameters such as `para->vRT` back to their original values, as well as freeing memory.

3.2.5 Counterexample Generation

We used the algorithm described in Section 2.5.3 for LTLK counterexample generation. The algorithm was already partially implemented in MCMAS, though there was a bug involving how the cases where it was not possible to loop back to a state in the first SCC were handled. This was illustrated by the example in Figure 2.7. We submitted a bug report and fixed the bug by restarting the algorithm from the state in which the "deadlock" occurred [57].

Once we fixed the aforementioned bug, implementing LTLK counterexample generation for an LTLK formula ϕ involved constructing $EG\top$ (line 2) and then finding a witness for $EG\top$ in the composition of the model and tableau, starting from a state in which $sat(\phi)$ does not hold:

```
1 // Compute EG True given eventualities are satisfied
2 modal_formula* spec = new modal_formula(11, new modal_formula(5, NULL));
3 // This enforces that the automaton is followed
4 modal_formula* val = ((modal_formula *) new_formula->obj[0]);
5 BDD sat_bdd = val->check_path_sat(para, index_map, next_index_map, false);
6 sat_bdd *= rules;
7 *state *= !sat_bdd; // select from non-accepting tableau states
8 bool successful_cex
9   = spec->build_cex(state, index, para, countex, idbdd, cextr);
```

(Note: `build_cex` is an existing function in `modal_formula` that constructs a witness.)

3.2.6 Comparison with MCK

MCK (introduced in Section 2.7.2) is an existing model checker for multi-agent systems. Although MCK does support verification of LTLK specifications (as discussed in Section 2.7.2), it is distributed as closed-source and there appears to be little published on the underlying theory of its LTLK implementation[2].

In addition to the existing *observational semantics* supported by both MCK and our tool, MCK also supports verification of fragments of LTLK over *clock semantics* (in which agents are aware of time) and *perfect recall semantics* (where agents remember all actions and past states)[3].

We did not carry out a performance comparison with MCK. However, given that both our extension and MCK use the tableau construction[4] to reduce the problem to one of CTLK model checking [6] and it was suggested that MCMAS tends to scale better than MCK over CTLK properties [65], it would be likely that MCMAS would scale better for LTLK properties as well.

[2]We had difficulty in finding any technical documentation for *how* MCK supports these specification languages, and others have apparently had such difficulties as well [70, 71].

[3]The formulas supported are the formulas which only use X, and $G\phi$ where ϕ itself is non-temporal.

[4]The part of the theory of MCK that was uncertain was how it handles epistemic modalities.

Chapter 4

Full Branching Time Epistemic Logic (CTL*K)

In this (relatively short) chapter we discuss how to perform model checking of Full Branching Time Logic (CTL*) with epistemic modalities. We apply the reduction of CTL* model checking to LTL model checking presented in [39], show that this still holds with epistemic modalities, and show that we still maintain the $O(2^{|\phi|} \times |\mathcal{I}|)$ time complexity of model checking LTL.

We then present how we extended our LTLK model checker (as described in Section 3.2) to handle CTL*K model checking and counterexample generation. The original work discussed in this chapter that was developed as part of this individual project is as follows:

1. **Extension of MCMAS to support CTL* semantics.** As discussed in Section 2.3.3, there are properties that can be expressed in CTL* that can be expressed in neither LTL nor CTL. Thus, this extension further expands the spectrum of properties which MCMAS is able to verify.

2. **Extension of MCMAS to support epistemic modalities for CTL* properties.** We extended our CTL* extension of MCMAS to support epistemic modalities. This allows us to support CTL*K in full.

4.1 Algorithm

4.1.1 Recursive Descent over Path Quantifiers

The reduction of the CTL* model checking problem to the LTL model checking problem is documented in [39]; we can similarly adapt it to handle CTL*K using a construction analogous to what was described in Section 3.1. Recall that if we build the tableau for a given path formula ϕ as described in Section 2.5.2 and compose it with the model, the states which satisfy $EG\top$ with the appropriate fairness constraints as well as $sat(\phi)$ are the ones from which there exists an infinite path satisfying $E\phi$. Since $A\phi = \neg E\neg\phi$ this is sufficient for model checking an *arbitrary* CTL*K specification.

To check if $\mathcal{I}, g \vDash \phi$ where ϕ is a CTL*K state formula, we proceed inductively on the structure of ϕ. The propositional cases are relatively simple; we focus in greater detail on the $E\phi$ case.

We first compute the set of states in which each top-level epistemic subformula $K_i\psi, E_\Gamma\psi, D_\Gamma\psi$

or $C_\Gamma\psi$ of ϕ, or path-quantified subformula $E\psi$ or $A\psi$ holds. (This may involve recursive computations, of course.) Similar to the LTLK algorithm, we again treat these as atomic propositions when computing the states in which ϕ holds. As before, formulas are finite and these quantifiers must thus necessarily have a finite nesting depth. For example, consider the CTL*K formula

$$\phi = A((p \vee E(FK_aq))U(XA(Gq)))$$

where p, q are atomic propositions. We can compute the set of states in which ϕ holds as follows:

1. First compute the set of states in which K_aq (in red) holds. This involves using the epistemic accessibility relation for agent a on $h(q)$.

2. Then compute the set of states in which $E(F(K_aq))$ (in blue) holds; we can use a modified version of the tableau algorithm as discussed in the first paragraph of this section. This is equivalent to finding the states where $E(Fr)$ holds for fresh atomic proposition r, where r holds in precisely the states with K_aq.

3. Then compute the set of states in which $A(Gq)$ (in green) holds; this is equivalent to the set of states in which the LTL formula Gq holds. We can use the tableau algorithm of Section 2.5.2 to find this (or, alternatively, rewrite this as $\neg E\neg(Gq)$).

4. Finally compute the set of states in which ϕ holds. This is equivalent to the set of states in which $A((p \vee s)U(Xt))$ holds, where s and t are fresh atomic propositions holding in precisely the states where $E(F(K_aq))$ and $A(Gq)$ hold respectively; the formula here is an LTL formula and we can use the tableau algorithm to find this set of states.

4.1.2 Complexity Analysis

This procedure for CTL*K preserves the worst-case time complexity of LTL model checking – that is, $O(2^{|\phi|} \times |\mathcal{I}|)$. We proceed via an argument similar to what was presented in Section 3.1.2.

As before, observe that $|\mathcal{I}|$ is constant, while $|\phi|$ is decreasing after each recursive call. Similarly, we define $f(|\phi|, |\mathcal{I}|)$ as the time taken to evaluate a CTL* formula ϕ with model size $|\mathcal{I}|$. Now, consider the work required when combining results:

- As before, assigning atomic propositions is $O(|\phi|)$ with a suitable data structure.

- At each step, we may either have to compute the states over which an epistemic modality holds, or compute the states over which a path formula holds. This requires time bounded by the same asymptotic complexity as that of the LTLK model checking algorithm, as discussed in Section 3.1.2.

- Alternatively, we may need to evaluate a propositional formula over the states of the model (e.g. for a state formula $p \wedge q$). This can be done in $O(|\phi||\mathcal{I}|)$ time; observe that each operator can be evaluated in $O(|\mathcal{I}|)$ time since this involves a constant number of unions, intersections or complements over a set of states bounded by size $O(|\mathcal{I}|)$, and there are at most $O(|\phi|)$ such operators.

We can thus apply Lemma C.1, giving us the result that this CTL*K model checking algorithm has time complexity of $O(2^{|\phi|} \times |\mathcal{I}|)$. However, our implementation uses an $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$ algorithm for LTLK, and so its time complexity is, by Lemma 3.1, also $O(2^{|\phi| \log |\phi|} \times |\mathcal{I}|)$. As before, we believe this is outweighed by the practical efficiency of symbolic approaches.

4.2 Implementation

At a high level, we needed to modify the following aspects of MCMAS:

- Add support for parsing formulae specified in CTL*K (Section 4.2.1).
- Implement the recursive descent approach over both epistemic modalities and path quantifiers (Section 4.2.2).
- Finally, add support for counterexample and witness generation. If a formula $A\phi$ is false, we want a counterexample trace; if a formula $E\phi$ is true, we want a witness trace (Section 4.2.3).

4.2.1 Expression Parsing

Similar to implementing LTLK model checking, the only section of MCMAS' implementation of the ISPL grammar that I needed to change was the `Formulae` section. We similarly require users to explicitly indicate that they are specifying formulae in CTL*; in particular, since CTL* subsumes both CTL and LTL this distinction is useful. The following changes were made to the grammar:

1. A new `CTLSPREFIX` terminal was added which is "`CTL*`". It may be possible that users have used this as an identifier, though it seems unlikely and it seems that in most cases a simple `sed` operation renaming said identifier would be able to fix such ISPL programs. Note that no other new tokens were needed, as "A" and "E" are already reserved because they are used when parsing CTL's *AU* and *EU* expressions.

2. The `formlist` nonterminal was extended with several additional rules allowing parsing of CTL*K formulae.

3. Two nonterminals, `ctlsformula` and `pathformula` were added, which have several production rules corresponding to the syntax of CTL*K state and path formulae, respectively.

   Notice that although LTLK formulae generally follow the CTL*K path formula syntax, we prefer to maintain separate nonterminals and rules for them, partly to help enforce consistency of specification languages within a formula as well as because a CTL*K path formula can also be a state formula. Unifying `pathformula` and `ltlformula` without further checks would allow erroneous constructions such as `LTL p or X (E q);` to be parsed.

4. The `epistemicprefix` and `gepistemicprefix` nonterminals again had to be duplicated, to force consistency of specification languages across epistemic modalities and avail the user of the requirement of respecifying this every time an epistemic modality is used.

5. As far as parsing LTLK formulae is concerned, while the production rules of `ltlformula` are kept separate from those of `pathformula`, the object returned from parsing an LTL formula ϕ is actually treated very similarly to a CTL*K formula of the form $A\phi$, apart from the `to_string()` method. We have added an A path quantifier with `op` value to 50. We have also added a new E path quantifier, with an `op` value of 51. (We decided not to merge the LTL root token with the A path quantifier, as this would affect the output presented to users.)

4.2.2 Recursive Descent

Instead of simply computing the top-level *epistemic* subformulae as discussed in Section 3.2.3, we change the relevant method (previously called `compute_epistemic_subformulae()`) to evaluate all top-level epistemic or path-quantified subformulae. As before, these are then treated as atomic propositions throughout the rest of the computation.

To avoid unnecessary negation, we have also changed the way in which we use the tableau construction. Calling `check_formula()` on a CTL* formula of the form $A\phi$ (as well as an LTL formula) rewrites said formula as $\neg E \neg \phi$ and checks the states in which the new formula holds. The tableau construction described in Section 3.2 is used in a largely unchanged form (other than the handling of further nested path quantifiers as atomic propositions) to check formulae of the form $E\phi$ up to and including the checking of $EG\top$ with fairness. Thereafter, since we want to find the states from which there exists an infinite path on which ϕ holds in the starting state, it is sufficient to find the states in the product satisfying both $EG\top$ and $sat(\phi)$, and then determining the corresponding states in the original model. This is implemented as follows:

```
// infinite_paths is the BDD with EG T + fairness
BDD sat_f = new_formula->check_ltl_sat(para, index_map, 0) * rules;
infinite_paths *= sat_f;

// Find the original states with E f
// (Logic to abstract away tableau states, as in section 3.2.4)

result = infinite_paths;
// (the product of result with the reachable states is returned)
```

4.2.3 Counterexample and Witness Generation

The algorithm described in Section 2.5.3 for LTL counterexample generation is also applicable to CTL* counterexample or witness generation. Consider that for a given path formula ϕ, the counterexample generation algorithm returns an infinite path on which ϕ is satisfied – in the case of LTL counterexample generation, we used this algorithm to search for infinite paths in the composition satisfying $\neg\phi$. For CTL*, we still find counterexamples for universally quantified path formulae in the same way. We also find *witnesses* demonstrating that an existentially quantified path formula holds, using the same algorithm, with a small change (in particular, we look for states in $sat(\psi)$ as opposed to states not in $sat(\psi)$). This is implemented as follows:

```
// EG True given eventualities are satisfied
modal_formula* spec = new modal_formula(11, new modal_formula(5, NULL));

// This enforces that the automaton is followed
modal_formula* val = ((modal_formula *) new_formula->obj[0]);
BDD sat_bdd = check_path_sat(val, para, index_map, next_index_map, false);
sat_bdd *= rules;
*state *= sat_bdd; // This is changed from the LTL algorithm!
bool successful_cex
  = spec->build_cex(state, index, para, countex, idbdd, cextr);
```

Our algorithm only does this with one level of depth – in other words, supposing we had a formula $E(G(\phi \wedge E(F(\psi))))$, the algorithm returns a witness for the outer formula, without identifying a witness for $E(F(\psi))$ at each node in the path. Computing nested witnesses is not difficult to add, but was not done in the interest of time (as well as readability of the output).

Chapter 5

Linear Dynamic Epistemic Logic (LDLK)

One of the shortcomings of LTLK and CTL*K is that they cannot express all ω-regular properties, such as "p holds in every other state" [99]. This limitation, which has been claimed to arise in practice [60] is addressed by Linear Dynamic Logic (LDL) which was introduced by Vardi in [96]. LDL has since sparked some research interest, with several analyses carried out into variants of the logic [35, 41, 98].

In this chapter we discuss our novel approach for carrying out symbolic model checking of Linear Dynamic Logic (LDL) with epistemic modalities (LDLK). We begin by reducing the LDLK model checking problem to the LDL model checking problem. We then adapt the construction in [41] to build an alternating automaton recognising the paths in which an LDL formula holds. We then employ the algorithm of [20] to construct a symbolic Büchi automaton, and compose it with the model of our interpreted system. We show that this construction runs in time exponential in the size of the formula, but linear in the size of the model.

We then discuss our concrete implementation of a symbolic model checking tool for LDLK specifications. We conclude with a discussion of several performance optimisations, both theoretical and practical, and an explanation of how we have implemented them in our tool. The original work discussed in this chapter that was developed as part of this individual project is as follows:

1. **Combination of epistemic logic and LDL**. To the best of our knowledge, LDL has not been applied in the context of epistemic logics before. We extend LDL with epistemic modalities, giving rise to LDLK, and give an algorithm that allows us to perform model checking of LDLK. We show that LDLK, like LDL, is PSPACE-complete.

2. **Critical set based alternating automaton construction for LDL.** We introduce a notion of critical sets, which improves on the construction in [41] in that it does not require us to consider every possible ϵ-path. This allows us to construct an alternating automaton recognising the paths in which a given LDL formula holds.

3. **Extension of MCMAS to support LDL.** We implement our critical set based alternating automaton construction for LDL, as well as the symbolic breakpoint construction of [20] to convert these alternating automata into equivalent symbolic Büchi automata. We then compose said symbolic Büchi automata with the model of our system, to verify

whether LDL properties hold. To the best of our knowledge, this is the first model checker for LDL.

4. **Extension of MCMAS to support LDLK.** We concretely implement our approach for handling epistemic modalities in our extension of MCMAS. This allows us to support verification of LDLK properties. Again, to the best of our knowledge, this is the first model checker for LDLK.

5. **Performance Optimisations for LDLK model checking.** We discuss how we minimise the ϵ-NFAs constructed for a given regular expression at each step. We introduce two further optimisations – *propositional shortcircuiting* and *conjunct separation*. These optimisations have helped our tool scale to larger state spaces and/or perform verification more quickly. Furthermore, we discuss practical implementation techniques used to compute the symbolic breakpoint construction efficiently (as far as BDDs are concerned). We have implemented all of these optimisations in our tool.

5.1 Algorithm

5.1.1 Alternating Automata

We first use the recursive descent approach as explained in Section 3.1.1 to handle epistemic modalities. Given an LDLK formula, we first evaluate any epistemic subformulae, and introduce fresh atomic propositions that hold in precisely the states in which said epistemic subformulae. For example, to model check the LDL formula

$$\langle (K_a(r)? + K_b(q)?); (K_b(q)? + K_b(p)?); \top \rangle K_a(r)$$

we can first determine the states in which $K_a(r)$, $K_b(q)$ and $K_b(p)$ each hold. We then define α, β and γ to hold in said sets of states, and model check

$$\langle (\alpha? + \beta?); (\beta? + \gamma?); \top \rangle \alpha \tag{5.1}$$

In this way, we reduce the problem of LDLK model checking to LDL model checking.

We can then proceed using the alternating automaton construction described in Section 2.6.3, constructing ϵ-NFAs for regular expressions as well as alternating automata as necessary. However, there is one notable issue with a concrete implementation of this construction – recall Equation 2.1 for the transition relation of automata for formulae of the form $\langle \phi \rangle \psi$, presented again below:

$$\rho(s, A) = \begin{cases} \rho_\psi(s, A) & s \in S_\psi \\ \rho_j(s, A) & s \in S_j, j \in \{1, \ldots, k\} \\ \bigvee_{s' \in S_r \setminus \{f_r\}} \bigvee_{\pi \in \Pi(s,s')} \bigvee_{t \in \rho_r(s', A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s^0_j, A) \right) & \\ \vee & s \in S_r \\ \bigvee_{\pi \in \Pi(s,f_r)} \left(\rho_\psi(s^0_\psi, A) \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s^0_j, A) \right) & \end{cases}$$

Consider the $s \in S_r$ case; we require quantification over all ϵ-paths from state s to s' in the upper disjunct, and over all ϵ-paths from s to f_r in the lower disjunct. This may potentially be unbounded since ϵ-NFAs may not be acyclic (for example, consider the ϵ-NFA for p^{**}). For the $\langle \phi \rangle \psi$ modality, it is sufficient to consider simple paths [41], though the maximum number

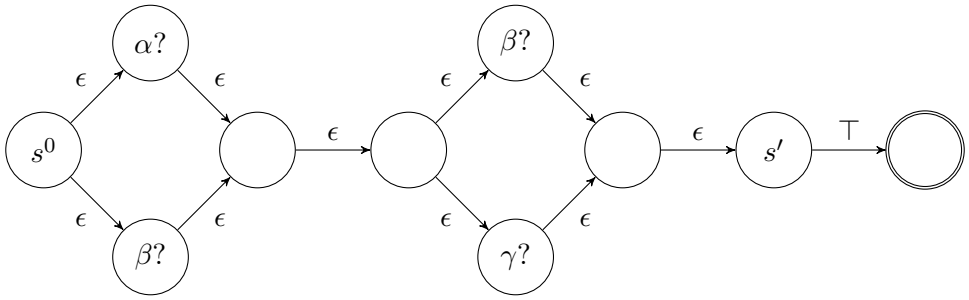


Figure 5.1: Marked ϵ-NFA for the modality $(\alpha? + \beta?); (\beta? + \gamma?); \top$

of simple paths in a graph with n nodes may still potentially be factorial[1]; simply storing all of these potential sets of markings may require factorial or exponential time and space. A possible approach could be to compute this relation on-the-fly, as is suggested in [41], though this tends to be difficult to combine with symbolic model checking [20].

5.1.2 Critical Sets

We thus introduce the notion of minimal and maximal *critical sets*, which allow us to reduce the number of paths we need to consider when computing the transition relation for the alternating automata we construct. Intuitively, the minimal critical sets refer to the smallest possible sets of tests that one needs to transition into on an ϵ-path from s to s'; the maximal critical sets refer to the largest possible sets of tests one needs to transition through. Viewed with the perspective of wanting to travel from s to s', they refer to the easiest (respectively, hardest) routes that one can take. We illustrate these for the formula 5.1 with an example, and then formally define them.

Consider the marked ϵ-NFA for the modality in formula 5.1, as shown in Figure 5.1. Intuitively, we can travel from the starting node s^0 to s', going through tests that pass, if we can pass β *or* we can pass both α and γ. Indeed, the minimal critical sets from s^0 to s' are $\{\{\beta\}, \{\alpha, \gamma\}\}$. If we had β and γ, we could travel through the first test for β and the γ test, but importantly, we didn't *need* γ (since we could have chosen to take the second β test instead).

Conversely, the *maximal* critical sets refer to the most "difficult" routes that one can choose. In this case, they are $\{\{\alpha, \beta\}, \{\alpha, \gamma\}, \{\beta, \gamma\}\}$. Notice that while $\{\alpha, \beta, \gamma\}$ is obviously sufficient, there is no route that requires all of them (and hence it is not a maximal critical set).

We now formally define *marking sets*, as a preliminary to formalising critical sets.

Definition 5.1. (Marking Sets) Let s, s' be two nodes in a marked ϵ-NFA $A = (\Sigma, S, s^0, \rho, f, m)$. $M \subseteq \text{range}(m)$ is a *marking set* from s to s' if there exists an ϵ-path $\pi = s_0, s_1, \ldots, s_n$ where $s_0 = s$, $s_n = s'$ and for every $i \in \{0, \ldots, n\}$, $m(s_i)$ is either undefined or in M. Furthermore, for every marking $\mu \in M$, there exists some s_i $(i \in 0, \ldots, n)$ such that $m(s_i) = \mu$. We say that a path $\pi = s_0, \ldots, s_n$ from s to s' *induces* a marking set M', where M' is the set of markings $\{k \mid \exists j \in \{0, \ldots, n\}.m(s_j) = k\}$.

Definition 5.2. (Critical Sets) Let s, s' be two nodes in a marked ϵ-NFA $A = (\Sigma, S, s^0, \rho, f, m)$. M is a *minimal critical set* of the markings range(m) from s to s', if:

[1]For a complete graph, this would be given by $\sum_{i=1}^{n} \left(i! \times \binom{n}{i}\right)$. A tighter bound might be possible for our automata.

- M is a marking set from s to s', and
- there does not exist $M' \subseteq \text{range}(m)$ such that $M' \subset M$ and M' is a minimal critical set.

The *maximal critical set* is defined dually; M is a *maximal critical set* of range(m) if:

- M is a marking set from s to s', and
- there does not exist $M' \subseteq \text{range}(m)$ such that $M \subset M'$ and M' is a maximal critical set.

We now show that to construct the transition relation for an automaton for formulae of the form $\langle\phi\rangle\psi$, instead of considering all possible sets of markings generated by the ϵ-paths from s to s', it suffices to consider only the minimal critical sets.

Theorem 5.1. Let π_1 be an ϵ-path from s to s' in a marked ϵ-NFA $A = (\Sigma, S, s^0, \rho, f, m)$, which was constructed for the regular expression ϕ. Let M_1 be the set of markings induced by π_1 and suppose M_1 is not a minimal critical set. Then the transition relation for the automaton for $\langle\phi\rangle\psi$ does not change even if we do not consider π_1.

Proof. We consider the case where $s' \neq f_r$ first. Since M_1 is a marking set from s to s', but not a minimal critical set, by Definition 5.2 there must exist some minimal critical set M such that $M \subset M_1$. Since M is a minimal critical set, it is a marking set and thus must have been induced by some path π_*. Now consider the transition relation ρ of the combined automaton. Clearly, the $s \in S_\psi$ and $s \in S_j$ cases are unaffected – we consider the $s \in S_r$ case. Clearly the second disjunct involving $\Pi(s, f_r)$ is unaffected. Now consider the first disjunct:

$$\bigvee_{s' \in S_r \setminus \{f_r\}} \bigvee_{\pi \in \Pi(s,s')} \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right)$$
$$= \bigvee_{s' \in S_r \setminus \{f_r\}} \left(\bigvee_{\pi \in \Pi(s,s') \setminus \{\pi_1, \pi_*\}} \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right) \vee Q \right)$$

where $Q = \bigvee_{\pi \in \{\pi_1, \pi_*\}} \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right)$. Now, we expand Q:

$$\bigvee_{\pi \in \{\pi_1, \pi_*\}} \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi)} \rho_j(s_j^0, A) \right)$$
$$= \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi_1)} \rho_j(s_j^0, A) \right) \vee \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j \in m(\pi_*)} \rho_j(s_j^0, A) \right)$$
$$= \bigvee_{t \in \rho_r(s',A)} \left(\left(t \wedge \bigwedge_{\theta_j \in m(\pi_1)} \rho_j(s_j^0, A) \right) \vee \left(t \wedge \bigwedge_{\theta_j \in m(\pi_*)} \rho_j(s_j^0, A) \right) \right)$$
$$= \bigvee_{t \in \rho_r(s',A)} \left(t \wedge \left(\bigwedge_{\theta_j \in m(\pi_1)} \rho_j(s_j^0, A) \vee \bigwedge_{\theta_j \in m(\pi_*)} \rho_j(s_j^0, A) \right) \right) \qquad \text{by distributivity}$$

Since $m(\pi_*) = M \subset M_1 = m(\pi_1)$, we can partition M_1 into the elements that are in M and the elements not in M. Thus we have:

$$\begin{aligned}
&\bigvee_{t\in\rho_r(s',A)} \left(t \wedge \left(\bigwedge_{\theta_j\in m(\pi_1)} \rho_j(s_j^0, A) \vee \bigwedge_{\theta_j\in m(\pi_*)} \rho_j(s_j^0, A) \right) \right) \\
= &\bigvee_{t\in\rho_r(s',A)} \left(t \wedge \left(\left(\left(\bigwedge_{\theta_j\in m(\pi_*)} \rho_j(s_j^0, A) \wedge \bigwedge_{\theta_j\in(m(\pi_1)\setminus m(\pi_*))} \rho_j(s_j^0, A) \right) \vee \bigwedge_{\theta_j\in m(\pi_*)} \rho_j(s_j^0, A) \right) \right) \\
= &\bigvee_{t\in\rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j\in m(\pi_*)} \rho_j(s_j^0, A) \right)
\end{aligned}$$

with the last step holding by the *absorption law* [48]. Finally, substituting this equivalent expression for Q back in the original formula and re-unifying the quantification over paths, we have

$$\begin{aligned}
&\bigvee_{s'\in S_r\setminus\{f_r\}} \left(\bigvee_{\pi\in\Pi(s,s')\setminus\{\pi_1,\pi_*\}} \bigvee_{t\in\rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j\in m(\pi)} \rho_j(s_j^0, A) \right) \vee Q \right) \\
= &\bigvee_{s'\in S_r\setminus\{f_r\}} \left(\bigvee_{\pi\in\Pi(s,s')\setminus\{\pi_1\}} \bigvee_{t\in\rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j\in m(\pi)} \rho_j(s_j^0, A) \right) \right)
\end{aligned}$$

Thus, as required, we need not consider π_1. The case for $s' = f_r$ is similar, involving distribution over the marking sets and absorption as before, though over the lower disjunct instead. We can also construct a dual argument for eliminating paths inducing non-*maximal* critical sets in the $[\phi]\psi$ case.

Hence, it suffices to consider the minimal (respectively, maximal) critical sets of markings when constructing the automata for $\langle\phi\rangle\psi$ (respectively, $[\phi]\psi$). Effectively, we reformulate ρ, the modified transition relation, where $MinCS(s, s')$ denotes the set of minimal critical sets from s to s', as:

$$\rho(s, A) = \begin{cases} \rho_\psi(s, A) & s \in S_\psi \\ \rho_j(s, A) & s \in S_j, j \in \{1, \ldots, k\} \\ \bigvee_{s'\in S_r\setminus\{f_r\}} \bigvee_{M\in MinCS(s,s')} \bigvee_{t\in\rho_r(s',A)} \left(t \wedge \bigwedge_{\theta_j\in M} \rho_j(s_j^0, A) \right) & \\ \vee & s \in S_r \\ \bigvee_{M\in MinCS(s,f_r)} \left(\rho_\psi(s_\psi^0, A) \wedge \bigwedge_{\theta_j\in M} \rho_j(s_j^0, A) \right) & \end{cases} \tag{5.2}$$

Similarly, for $[\phi]\psi$, where $MaxCS(s, s')$ denotes the set of maximal critical sets from s to s', we have:

$$\rho(s, A) = \begin{cases} \rho_\psi(s, A) & s \in S_\psi \\ \rho_j(s, A) & s \in S_j, j \in \{1, \ldots, k\} \\ \bigwedge_{s'\in S_r\setminus\{f_r\}} \bigwedge_{M\in MaxCS(s,s')} \bigwedge_{t\in\rho_r(s',A)} \left(t \vee \bigvee_{\theta_j\in M} \rho_j(s_j^0, A) \right) & \\ \wedge & s \in S_r \\ \bigwedge_{M\in MaxCS(s,f_r)} \left(\rho_\psi(q_\psi^0, A) \vee \bigvee_{\theta_j\in M} \rho_j(s_j^0, A) \right) & \end{cases} \tag{5.3}$$

We can verify that the number of critical sets is sub-exponentially bounded in the size of the ϵ-NFA, as opposed to factorial.

Theorem 5.2. Suppose we have a marked ϵ-NFA with n nodes – the set of nodes is S. Then, the maximal number of minimal critical sets (and also the maximal number of maximal critical sets) is given by

$$\binom{n}{\lfloor \frac{n}{2} \rfloor}$$

Proof. First observe that, by definition, if M_1, M_2 are both minimal or maximal critical sets, then we cannot have $M_1 \subset M_2$. We formulate this problem in terms of a graph problem on the *Hasse diagram* for the power-set of S, $\mathcal{P}(S)$ under the relation $\subseteq$, taking definitions from [90].

Definition 5.3. (Immediate Predecessors and Successors) Let R be a partial order on a set S, and $a, b \in S, a \neq b$. a is an *immediate predecessor* of b under R if $(a, b) \in R$, and there does not exist any $c \neq a, b$ with $(a, c) \in R$ and $(c, b) \in R$. b is an *immediate successor* of a if a is an *immediate predecessor* of b.

Definition 5.4. (Hasse Diagram) A *Hasse diagram* is a compact way of representing a partial order between elements in a set S as a directed graph. The nodes of the graph are the elements of S. Instead of representing all pairs of elements $(a, b) \in S \times S$ in the partial order, we *only* include an edge (a, b) in the graph if a is an immediate predecessor of b.

As discussed, we consider the Hasse diagram for $\mathcal{P}(S)$ under $\subseteq$. Suppose our set of minimal (or maximal) critical sets is Q. Then, there cannot be a path between any 2 elements $q_1, q_2 \in Q$, $q_1 \neq q_2$ (if there was, then $q_1 \subset q_2$, which is not allowed). This is equivalent to identifying that over every path from $\emptyset$ to each $q_i \in Q$, and from q_i to S, no other $q_j \in Q (j \neq i)$ is also on the path.

We proceed by an inductive argument over the depth of nodes with respect to $\emptyset$. Let L be the set of nodes at depth i, $i < \lfloor \frac{n}{2} \rfloor$ and suppose there are no nodes at depths strictly less than i. We show that if Q respects the subset property, then it is always possible to replace L with immediate successors of L (i.e. depth $i + 1$) while preserving the subset property – the other sets are not affected because the new sets introduced are supersets of the old sets (which were not subsets of the other sets), and thus cannot be subsets of the other sets. It remains to show that choosing a set of immediate successors is possible.

Consider an arbitrary subset of L, L' and the immediate successors of L', R. Clearly, these form a bipartite graph. The nodes in L' must have $(n - i)$ immediate successors, since it is possible to add any of the $(n - i)$ elements not in the current set to reach an immediate successor. The nodes in R, on the other hand, can have *at most* $i+1$ predecessors (which would be obtained by removing each element). Since the total degrees of the nodes from each set must be the same,

$$|R| \geq \frac{(n - i)|L'|}{(i + 1)}$$

Since $i < \lfloor \frac{n}{2} \rfloor$ by construction, but i and $\lfloor \frac{n}{2} \rfloor$ are integers, we have $i \leq \frac{n}{2} - \frac{1}{2}$. Then,

$$\begin{aligned} i &\leq \frac{n}{2} - \frac{1}{2} \\ 2i &\leq n - 1 \\ i + 1 &\leq n - i \end{aligned}$$

$$\frac{n-i}{i+1} \geq 1$$
$$\therefore |R| \geq |L'|$$

Hence, since we selected an arbitrary subset of L, we conclude that for every subset of L, there are at least as many immediate successors. Now, consider Hall's theorem for matching in bipartite graphs (as may be found in [33]):

Theorem 5.3. (Hall's Theorem) Suppose G is a bipartite graph, with bipartite sets L and R. For every set $L' \subset L$, let $N(L')$ be the set of nodes in R that are connected to some node of L. Then, there is a matching that covers L if and only if $|L'| \leq |N(L')|$ for all subsets L' of L.

Thus, there exists a matching between L and its immediate successors that covers L. Applying the above result and working forwards from a depth of 0, for any set of minimal (or maximal) critical sets we can propagate elements at depths less than $\lfloor \frac{n}{2} \rfloor$ to depth $\lfloor \frac{n}{2} \rfloor$.

We use a similar argument in the other direction. Let R be a set of nodes at depth i, $i > \lfloor \frac{n}{2} \rfloor$ and suppose there are no nodes at depths strictly greater than i. We seek to show that if Q respects the subset property, then it is always possible to replace R with as many immediate predecessors of R. The other sets are not affected as the existing sets were not subsets of sets in R, and the new sets introduced are subsets of the sets in R. We now consider a subset of R, R' and its immediate predecessors L. They form a bipartite graph, as before; the nodes in R' have i immediate successors since it is possible to delete any of the i elements from each node, while the nodes in L can have at most $(n-i)$ successors each (since there are only $(n-i)$ more elements that could be added). As before, the total degrees of the nodes from each set must be the same, so

$$|L| \geq \frac{(i+1)|R'|}{n-i}$$

Since n is a positive integer, $\lfloor \frac{n}{2} \rfloor \geq \frac{n}{2} - \frac{1}{2}$. Thus, $i > \frac{n}{2} - \frac{1}{2}$, and so

$$i \geq \frac{n}{2} - \frac{1}{2}$$
$$2i \geq n - 1$$
$$i + 1 \geq n - i$$
$$\frac{i+1}{n-i} \geq 1$$
$$\therefore |L| \geq |R'|$$

Since we selected an arbitrary subset of R, for every subset of R there are at least as many immediate predecessors. Thus again, by Hall's theorem there exists a matching between R and its immediate predecessors that covers R. Thus, we can similarly propagate elements at depths greater than $\lfloor \frac{n}{2} \rfloor$ back to $\lfloor \frac{n}{2} \rfloor$. This would lead to a contradiction if there were more than $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ critical sets, since there are only $\binom{n}{\lfloor \frac{n}{2} \rfloor}$ possible elements at depth $\lfloor \frac{n}{2} \rfloor$. Note that the bound is achievable since we can select all of the elements at depth $\lfloor \frac{n}{2} \rfloor$ as critical sets.

Hence, this does reduce the number of test sets we need to consider for each node from potentially factorial to sub-exponential (we can bound it more tightly to $O(n^{-1/2}2^n)$ [42]). Note that we do not need to store all of the test sets explicitly; we can in fact store a Boolean combination of the states to be transitioned into.

5.1.3 Finding Critical Sets

Finding critical sets can be done by a variant of breadth-first search (BFS) over the ϵ-closure of each node in the alternating automaton; we augment the BFS to keep track of the current node as well as test sets used along the current path. We also prune paths early if we can be sure that exploring them yields no benefit. For example, to find the minimal critical sets for all states in the ϵ-closure of s, we can proceed as follows (the algorithm below is expressed in pseudocode):

```
1  let closure be a map<node, set<set<test>>>
2  add a pair (s, {}) to queue Q
3  while Q is not empty
4    node curr = Q.front.first();
5    set<test> test_set = Q.front.second();
6    remove front of Q
7
8    if (closure[curr] is null)
9      closure[curr] = {test_set}
10   else if (closure[curr] contains a subset of test_set)
11     continue
12   else if (closure[curr] contains a superset of test_set)
13     closure[curr] = (closure[curr] union {test_set}) - {superset}
14   else
15     closure[curr] = closure[curr] union {test_set}
16
17   // Note: If we hit continue above, this part does not run.
18   for each outgoing edge of curr
19     if (curr is an epsilon transition)
20       add (endpoint of transition, test_set union {marking at endpoint}) to Q
```

At each step in the BFS, we check if the set of tests required to reach a node is a subset or superset of a known set (the closure) for that node. If we know a set that is a subset of the current test set, then any more tests along this path cannot be part of a minimal critical set, and we thus stop search along this path. The maximal case is similar, except we prune the search if a known set is a superset of our current set.

Based on experimental results, this step appears to require time exponential in the size of our alternating automaton, which is linear in the size of our formula [41]. Thus finding these sets appears to take time exponential in the size of the formula. We formalise a bound:

Lemma 5.1. In the minimal critical set algorithm, each test set does not travel down the same edge of the automaton more than once.

Proof. Suppose some test set T is propagated down some edge e. Since markings along edges do not change, suppose e has marking m; we reach line 17 of the algorithm with the set $T' = T \backslash \{m\}$. Consider that after T' is seen for the first time, either T' is already in the closure of the current node, or we have some alternative set $T'' \subset T'$ (if another set triggered line 13) in the closure. Thus the closure must contain a subset of T', which means we must have reached the continue statement on line 11. This means we do not reach line 17 with T' a second time. Hence, the same test set never travels down the same edge more than once, as required.

The proof for the maximal critical set algorithm is similar. There are $O(n^2)$ edges and $O(2^n)$ possible test sets; hence, using Lemma 5.1 we have at most $O(n^2 2^n)$ loop iterations. Furthermore, using suitable data structures (e.g. from [83]), the queries in lines 10, 12 and 14 can be answered in amortised linear time, yielding overall time complexity bounded by $O(n^3 2^n) = O(2^{n \log n})$. We suspect that the actual bound may be tighter, but this is sufficient for our purposes.

An alternative construction which would work for finding *minimum* critical sets could involve computing a topological ordering of a subgraph of the automaton graph which has all cycles removed while still remaining connected. The topological sort process involves augmenting depth-first search with suitable bookkeeping, and should run in time linear in the number of nodes and edges [33]. Thereafter, we can consider one node at a time, and take the union over the test sets built from the previous nodes intersected with the label at the current node (effectively, using a dynamic programming approach).

We did not focus on alternative search algorithms owing to limited development time, given that the computation time required for this step as observed in practice (as described in Section 5.4) is relatively insignificant.

5.1.4 Symbolic Breakpoint Construction and Model Composition

We thus can compute a representation of the transition relation for an alternating automaton representing an arbitrary LDL formula ϕ. This allows us to use the construction presented in [20] and discussed in Section 2.6.4 to build a symbolic structure A_ϕ (effectively, a symbolic Büchi automaton) that accepts a given infinite path if and only if it would have been accepted by our original alternating automaton. By composing this symbolic structure for ϕ, A_ϕ, and our model $\mathcal{I}$ together (taking the product of their states, and ensuring that they synchronise in terms of when atomic propositions hold, what transitions are allowed *etc.*), we can check whether there exists an infinite path on which ϕ holds by checking whether there exists a fair path in the model $\mathcal{I}$ on which A_ϕ also accepts.

Then, recalling the semantics of LDL over states from Definition 2.21, we have that for some state $g \in G$,

$$\begin{aligned}
\mathcal{I}, g \vDash \phi &\leftrightarrow \mathcal{I}, \pi \vDash \phi \text{ for all paths } \pi \text{ starting at } g \\
&\leftrightarrow \text{it is not the case that (not } (\mathcal{I}, \pi \vDash \phi \text{ for all paths } \pi \text{ starting at } g)) \\
&\leftrightarrow \text{it is not the case that } (\mathcal{I}, \pi \nvDash \phi \text{ for some path } \pi \text{ starting at } g) \\
&\leftrightarrow \text{it is not the case that } (\mathcal{I}, \pi \vDash \neg\phi \text{ for some path } \pi \text{ starting at } g)
\end{aligned}$$

Thus, if we want to find the states in which an LDL formula ϕ holds, we can proceed by explicitly building an alternating automaton for $\neg\phi$ and then using the symbolic construction to convert it into a symbolic Büchi automaton. We then compose it with the model $\mathcal{I}$ and check for infinite paths; the states returned by this process are those for which $\mathcal{I}, \pi \vDash \neg\phi$ for some path π starting at g. Thus, taking the complement of this set gives us the states for which $\mathcal{I}, \pi \vDash \phi$ for every path π starting at g i.e. the states satisfying ϕ.

5.1.5 Complexity Analysis

Following the steps we have presented above, we now outline how we can determine the states where $\mathcal{I}, g \vDash \phi$ and the time complexity of each step. Let the time complexity of our algorithm be $T(|\phi|, |\mathcal{I}|)$.

1. Scan ϕ for epistemic modalities. Compute the states in which said epistemic modalities hold, and introduce appropriate fresh atomic propositions to replace them. Supposing we have n epistemic modalities and each has size $k_0, k_1 \ldots k_n$ (all of which are necessarily smaller than $|\phi|$), this step requires time $\sum_{j=0}^{n} T(|k_j|, |\mathcal{I}|) + O(|\phi|)$.

2. Construct the formula $\neg\phi$ (after step 1), and convert $\neg\phi$ to negation normal form (Definition 2.43). This step may require time $O(|\phi|)$.

3. Construct an explicit alternating automaton corresponding to $\neg\phi$ (now in negation normal form). Note that compared to our approach for LTL or CTL* where we avoided explicitly building the Büchi automaton, explicitly constructing an alternating automaton here is acceptable because the alternating automaton is linear in the size of the formula (it has $O(|\phi|)$ many states). The time required for this construction is dominated by the step of finding critical sets. This step, as shown in Section 5.1.3, may take $O(2^{|\phi| \log |\phi|})$ time in the worst case.

4. Convert the alternating automaton from step 3 into a symbolic Büchi automaton. This step generates a symbolic Büchi automaton, though with an exponential blowup in the state space [20]. Our automata have $O(|\phi|)$ states, so in the worst case this step may require $O(3^{|\phi|})$ time.

5. Compose our symbolic Büchi automaton with the interpreted system $\mathcal{I}$, and check $EG\top$ with suitable fairness constraints. This step involves CTL model checking with fairness, which is linear in the size of the model, formula, and number of fairness constraints [13]. The size of the model is $O(3^{|\phi|} \times |\mathcal{I}|)$, the formula size is constant and there is one fairness constraint for the symbolic automaton. Hence, the time complexity of this step is $O(3^{|\phi|} \times |\mathcal{I}|)$.

6. Take the complement of the set of states returned in step 5 (since that was the set of states from which there existed an infinite path with $\neg\phi$). This step runs in $O(|\mathcal{I}|)$ time.

We claim the overall time complexity (that is, the sum over all steps) is $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$. To prove this, we need a slightly expanded version of Lemma 3.1 which accounts for the possibility of either of the terms dominating.

Lemma 5.2. Suppose that for $i \in 0, \ldots, k$ we have $f(y_i, n) = O(\max(2^{y_i \log y_i}, n3^{y_i}))$ and $0 \leq y_i < x$. Further suppose that $f(x, n) = \sum_{i=0}^{k} f(y_i, n) + O(\max(2^{x \log x}, n3^x))$. Then $f(x, n) = O(\max(2^{x \log x}, n3^x))$.

Proof. From the definition of big-O notation we have for each i, $f(y_i, n) \leq c_i \max(2^{y_i \log y_i}, n3^{y_i})$ for some positive c_i. Observe that because $y_i < x$, $2^{y_i \log y_i} < 2^{x \log x}$ and $3^{y_i} < 3^x$. We have

$$
\begin{aligned}
f(x,n) &= \left(\sum_{i=0}^{k} f(y_i, n)\right) + O(\max(2^{x \log x}, n3^x)) \\
&\leq \left(\sum_{i=0}^{k} c_i \max(2^{y_i \log y_i}, n3^{y_i})\right) + O(\max(2^{x \log x}, n3^x)) && \text{by above assertion} \\
&\leq \left(\sum_{i=0}^{k} c_i \max(2^{y_i \log y_i}, n3^{y_i})\right) + c \max(2^{x \log x}, n3^x) && \text{for some positive } c \\
&\leq \left(\sum_{i=0}^{k} c_i \max(2^{x \log x}, n3^x)\right) + c \max(2^{x \log x}, n3^x) \\
&= \left(\left(\sum_{i=0}^{k} c_i\right) + c\right) \max(2^{x \log x}, n3^x)
\end{aligned}
$$

Thus the result follows. This is (singly) exponential in the size of the formula, but linear in the size of the model, which is in line with the LTL and CTL*K algorithms.

5.2 PSPACE-Completeness of LDLK

Theorem 5.4. LDLK model checking is PSPACE-complete (provided the model $\mathcal{M}$ is given explicitly).

Proof. PSPACE-hardness follows from a straightforward reduction of LDL model checking, which is known to be PSPACE-complete [41], to LDLK model checking. Supposing the LDL and LDLK model checking problems involve, for a given instance $(\mathcal{M}, s, \phi)$ determining if $\mathcal{M}, s \vDash \phi$, we have that $(\mathcal{M}, s, \phi)$ is a yes-instance of LDL model checking iff $(\mathcal{M}, s, \phi)$ is a yes-instance of LDLK model checking, since ϕ does not contain epistemic modalities and the semantics of LDLK apart from epistemic modalities match precisely that of LDL.

We now show PSPACE membership, by showing that our LDLK algorithm will operate in polynomial space if the underlying LDL model checking algorithm operates in polynomial space. Such an algorithm exists, since LDL model checking is in PSPACE. The maximum possible depth of recursion is bounded by the length of the formula, and hence polynomial in the input size (so we can use, say, a stack to remember where we are in the recursion). Furthermore, at each step there are at most polynomially many recursive calls to be made (again, bounded by the length of the subformula currently being analysed). The total number of expressions which have epistemic modalities as their principal connective is thus polynomial. We can compute the states in which each of these epistemic subformulae holds, by using the LDL model checking algorithm (which runs in polynomial space) and applying the epistemic accessibility relation (which is a subset of the Cartesian product of the states of $\mathcal{M}$, and hence polynomial in the input size). We can also store the result of each of these recursive calls (even though there might not be a need to) because they return a subset of the states, which is polynomially bounded in the size of $\mathcal{M}$ (since it was given explicitly). Thus, the entire procedure can operate in polynomial space, thus showing PSPACE membership.

Combining the above two results shows that LDLK model checking is PSPACE-complete, like LDL model checking.

5.3 Implementation

5.3.1 Overall Solution Architecture

Owing to the greater complexity of the construction for LDL as opposed to LTL and CTL*, we introduce several objects that collaborate in our implementation:

- `ldl_modality`, which encapsulates a regular expression. These objects are created when parsing the regular expressions inside dynamic modalities. They also support compilation into marked ϵ-NFAs.
- `automaton_node` and `automaton_edge`, which represent the nodes and edges of the ϵ-NFAs and the alternating automata.
- `finite_nfa`, which represents a marked ϵ-NFA and supports computation of minimal and maximal critical sets. We construct these objects when constructing the alternating automata for an LDL formula of the form $\langle\phi\rangle\psi$ or $[\phi]\psi$.
- `dynamic_automaton`, which represents an alternating automaton constructed for an LDL formula and supports compilation into a symbolic Büchi automaton for composition with the model being checked.

- `modal_formula`, which coordinates the construction, use and destruction of the aforementioned classes.

At a high level, we modify or extend MCMAS to perform the following:

- Parse regular expressions and LDLK formulae (Section 5.3.2).
- Construct marked ϵ-NFAs and support finding critical sets in these ϵ-NFAs (Section 5.3.3).
- Construct a `dynamic_automaton` – an alternating automaton for the LDLK formula to be checked (Section 5.3.4).
- Execute the symbolic breakpoint construction (Section 5.3.5).
- Construct the automata and perform the composition correctly when checking an LDLK `modal_formula` (Section 5.3.6).
- Generate counterexample paths for LDLK formulae that do not hold (Section 5.3.7).

5.3.2 Expression Parsing

As before, the only section of MCMAS' implementation of the ISPL grammar we needed to change was the `Formulae` section; we would need to add the ability for users to specify LDL formulae here. The following changes were made to the grammar:

1. We prefer users to explicitly state which specification language they are using, as previously discussed in Section 3.2.1. We added a new terminal, `LDLPREFIX` which is "`LDL`".
2. The `formlist` nonterminal, which is used to parse lists of formulae (line 55 in Definition 2.45), was extended with several production rules allowing parsing of `ldlformula`s.
3. The `ldlformula` nonterminal was added, which has several production rules corresponding to the syntax of LDL formulae. We reused the existing terminals for "`<`" and "`>`" which are used for comparison of fixed-range integers, but added two new terminals for "`[`" and "`]`".
4. The `ldlmodality` nonterminal was added, which has several production rules corresponding to the syntax of the dynamic modalities in LDL. We reused the existing terminals for "`+`" and "`*`", which are already used to support arithmetic over fixed-range integers, as well as "`;`" which is normally used to terminate statements, but had to add a new terminal for "`?`" (which is used to represent tests).
5. Finally, we added suitable versions of the `epistemicprefix` and `gepistemicprefix` nonterminals, to support LDL specifications while avoiding requiring the user to re-specify that he was writing formulae in LDL.

In addition, we suitably extended `syntaxcheck.cc` to ensure that all atomic propositions appearing in each formula as well as in each dynamic modality were well defined. We also added a further check to verify that the formulae in dynamic modalities (apart from tests) were propositional formulae (i.e. ensuring that syntactically malformed formulae such as $\langle\langle p\rangle q\rangle r$ were rejected), following the syntax of LDL presented in Definition 2.20.

The parsing step creates a `modal_formula` object for each LDL formula; at the root, these have an `op` value (used internally within MCMAS to distinguish operators) of 80, using a technique

similar to our construction for LTL. We also used `op` values of 70 and 71 to refer to formulae of the form $\langle\rho\rangle\phi$ and $[\rho]\phi$, respectively. Naturally, we had to extend `modal_formula` with an additional constructor that accepts a `ldl_modality` as its first operand, and a `modal_formula` as its second operand; the existing constructors only supported varying numbers of `modal_formula` operands, or an `atomic_proposition` argument.

The `ldl_modality` objects representing regular expressions are also created during parsing; similar to `modal_formula`, we internally distinguish them using operator values. The class also has multiple constructors, corresponding to the varying cases:

```
class ldl_modality:public Object
{
  /* 0 = p
   * 1 = p?
   * 10 = m1;m2
   * 11 = m1+m2
   * 20 = m1*
   */
  unsigned char op;
  Object* obj[2]; // arguments
public:
  // One argument, modal formula case
  ldl_modality(unsigned char o, modal_formula * m1);

  // One argument, modality case (star)
  ldl_modality(unsigned char o, ldl_modality * m1);

  // Two modality arguments (+ and *)
  ldl_modality(unsigned char o, ldl_modality * m1, ldl_modality * m2);

  // Construct the marked eps-NFA
  finite_nfa* build_regex_nfa();

  // omitted: utility methods
};
```

5.3.3 ϵ-NFAs and Critical Sets

The aforementioned `ldl_modality` class has a `build_regex_nfa()` method, which constructs the marked ϵ-NFA for that dynamic modality. This is represented by the `finite_nfa` class, which maintains a start state, a set of accepting states as well as a map of states to modal formulae. We use a standard adjacency list representation for the automata, since we expect said automata to be fairly sparse:

```
struct automaton_node
{
  vector<automaton_edge*>* transitions;
};

struct automaton_edge
{
  automaton_node* endpoint;
  bool is_epsilon;
  modal_formula* transition_label;
};
```

The `finite_nfa` class itself is implemented with several factory methods that follow Thompson's construction (we have one method for each type). The class also features two key methods

– `epsilon_closure_minimal` and `maximal`, which take a starting node and return all nodes in the ϵ-closure of said starting node, along with their minimal or, respectively, maximal critical sets. We simply use an explicit set of sets of modal formulae here, as we find in practice that this is not a performance bottleneck (even though, as noted in Section 5.1.2, this could potentially use exponential space in the worst case).

The `epsilon_closure` methods directly implement the breadth-first search algorithm put forth in Section 5.1.3. We directly iterate through the set of sets in the closure, looking for a subset; this may potentially require exponential time in the worst case, as compared to the possible guarantee of amortised linear time if we use a set-trie [83], but we have found this to not be a performance bottleneck in practice as well.

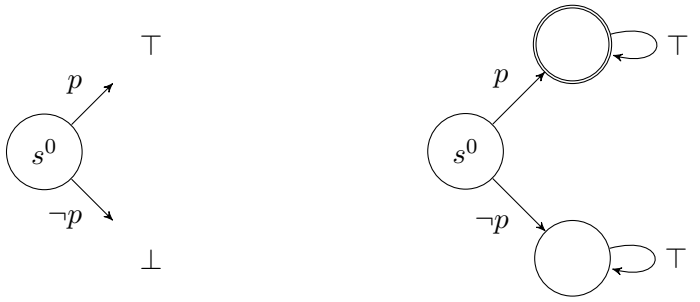
5.3.4 Alternating Automaton Construction

The `dynamic_automaton` class is used to represent an alternating automaton. We highlight its key fields and methods below:

```
class dynamic_automaton:public Object
{
  map<automaton_node*, BDD>* transition_mapping;
  automaton_node* initial_state;
  set<automaton_node*> accepting_states;
  map<automaton_node*, BDD> state_map;

  // omitted: private constructor populating the above fields

public:
  // Constructs a dynamic automaton for the given formula and substitutions
  static dynamic_automaton* construct_dynamic_automaton(
    modal_formula* ldl_formula, bdd_parameters* para,
    map<modal_formula*, BDD>* epistemic_valuation);

  // Return states from which the automaton has an accepting path
  BDD check_dynamic_automaton(bdd_parameters* para);

  // Construct a witness path.
  // Named as such for consistency
  bool build_cex(BDD* state, unsigned int index, bdd_parameters* para,
     vector<vector<int>*>* countex, map<int,BDD*>* idbdd,
     vector<vector<transition*>*>* cextr);

  // omitted: private utility methods
};
```

The contents of the fields are fairly self-explanatory, apart from the `state_map`; this maps nodes of the automaton to BDD variables, which are used for efficient symbolic manipulation of the transition mapping (especially when combining sub-automata) as well as compilation into symbolic Büchi automata with the symbolic breakpoint construction (discussed in greater detail in Section 5.3.5).

We use the *factory pattern* to create objects of this class, owing to the complexity of construction; given an `ldl_formula` in negation normal form, a pointer to a struct `bdd_parameters` which represents data about how MCMAS is using BDDs, and a mapping of epistemic subformulae to the states in which they hold, the `construct_dynamic_automaton` method builds an alternating automaton that accepts precisely the paths on which the input `ldl_formula` holds.



(1) Standard construction (2) Alternative, implemented construction

Figure 5.2: Alternative implementations of alternating automata for the atomic proposition p.

`construct_dynamic_automaton` recurses over the formation tree of the input `ldl_formula`, explicitly constructing the states of the alternating automaton (which we find acceptable, given that it is only linear in the size of the formula – note that transitions are symbolically represented). We proceed by discussing the various cases in turn.

Base Cases

The base cases involve atomic propositions, epistemic modalities (recall that these are treated as fresh atomic propositions holding in specific states), and negations of these (we have a precondition that the formula is in negation normal form). We need to construct automata that transition to $\top$ if the proposition holds (or its negation for negated atoms), and $\bot$ otherwise; this is pictured in item (3) in Figure 2.10.

Our implementation is slightly different; instead of handling $\top$ and $\bot$ as special cases, we introduce trap states which have the same effect, in that a transition into these trap states is always (for $\top$) or never (for $\bot$) accepting. We thus introduce *three* states; a `start` state, an `accept` state and a `reject` state. The transition mapping is defined such that the accept and reject states always transition *only* to themselves, and the start state transitions into the appropriate state depending on whether or not the relevant proposition holds (negated or otherwise). This is illustrated in Figure 5.2, and implemented as follows:

```
1 transition_mapping->insert(
2     make_pair(start_node,
3               (atomic_states * state_map[accept_node]) +
4               (!atomic_states * state_map[reject_node]))
5 );
6 transition_mapping->insert(make_pair(accept_node, state_map[accept_node]));
7 transition_mapping->insert(make_pair(reject_node, state_map[reject_node]));
```

(Note that if a proposition is negated, `atomic_states` is already negated before line 1 in this listing.)

Finally, we mark the accept state (and *only* the accept state) as an accepting state of the alternating automaton, and invoke the constructor.

Propositional Cases

For the propositional cases ($\wedge$ and $\vee$; we also support $\rightarrow$ for usability reasons), we first recursively construct automata for the two subformulae. We then take the union of the two sub-

automata. Further to that, we introduce a fresh starting state `new_start` which can transition into either of (or both of) the sub-automata as necessary. The new starting state can transition into the successors of the initial state of the sub-automata, depending on which propositional case we are dealing with:

```
BDD first_successors
  = automaton1->transition_mapping->find(automaton1->initial_state)->second;
BDD second_successors
  = automaton2->transition_mapping->find(automaton2->initial_state)->second;

if (op == 1) {
  new_start_relation = first_successors * second_successors; // AND
} else if (op == 2) {
  new_start_relation = first_successors + second_successors; // OR
} else if (op == 4) {
  new_start_relation = !first_successors + second_successors; // IMPLIES
}
```

We then mark `new_start` as the *only* starting state. (Note that we *cannot* necessarily remove either of the original starting states, since the sub-automata may not necessarily be acyclic.)

Modal Cases

We outline the implementation of our construction for $\langle\rho\rangle\phi$ or $[\rho]\phi$. Note that our top-level `modal_formula` has two arguments in this case – we have an `ldl_modality` for ρ, and we have a `modal_formula` for ϕ.

We first recursively construct an automaton for ϕ. We then construct the marked ϵ-NFA for ρ (using the `build_regex_nfa()` method introduced in Section 5.3.3). Next, we iterate through the states of the marked ϵ-NFA; we iteratively construct the updated transition relation for each node, as well as (recursively) build automata for the tests (or their negations, for the box case). For each state, we first retrieve its minimal or maximal ϵ-closure using the methods described in Section 5.3.3. We then iterate over every node in the closure and construct the transition relation as indicated in Equation 5.2 for the diamond case, or Equation 5.3 for the box case using BDDs. Whenever we encounter a test θ, we recursively construct an automaton for it (diamond case) or its negation (box case), and add that automaton's states and transition relation to our current automaton (by taking the union).

Finally, we invoke the constructor. The starting state of the overall automaton is the starting state of the automaton for ρ. It is worth noting that *specifically* for the box modality, there is an additional step of making every node in the automaton for ρ accepting (since we are quantifying over the cases after a successful match, not asserting the existence of a successful match).

We have thus covered all of the cases and outlined how, in practice, we construct a representation of an alternating automaton (a `dynamic_automaton`) from a `modal_formula` after suitable preprocessing (for the epistemic modalities, and conversion to negation normal form).

5.3.5 Symbolic Breakpoint Construction

We implemented a symbolic version of the breakpoint construction from [20], discussed in Section 2.6.4, in the `dynamic_automaton` class. The construction featured in the paper is somewhat more complex as it introduced an additional optimisation for parts of the alternating automata that are very weak; for simplicity of implementation, as well as because the experimental results

were not conclusive, we use the "basic" breakpoint construction as outlined in Section 2.6.4. We implement this as follows:

- We first create the BDD variables required by the construction. For each state s in the alternating automaton:
  - If s is accepting, we need to consider whether it is in the current set L, and whether we will be transitioning to it in the next state (recall that we require two copies of the variables for symbolic model checking, as explained in Section 2.4.3). We thus create *one* additional BDD variable for the next states L' (we reuse the BDD variables that were used when constructing the transition relation for the alternating automaton for checking whether states are currently in L). Let the variables for s be s_L and s'_L.
  - Otherwise, we create *three* additional BDD variables corresponding to whether s is in the current set L, whether it is in R, as well as the next states for each of these variables. (As before, we reuse the existing BDD variables for checking whether states are currently in L). In this case, we refer to the variables for s as s_L, s_R, s'_L and s'_R.

  We thus declare three additional maps of automaton nodes to new BDD variables; these maps are referenced throughout our construction. Let us define S_L to be the original set of variables as found in the state map and S_R to be the set of variables checking whether states are in the current R set. Furthermore, let S'_L and S'_R be the next versions of these sets of variables.

- We want our symbolic structure to accept a path if and only if the alternating automaton would also accept it. This is satisfied if R is perpetually empty; that is, we introduce one fairness constraint $F' = \bigwedge_{s \in S \setminus F} \neg s_R$.

- We then set up the transition relation. The authors of [20] decompose this into three parts, which we follow in our adaptation for our implementation:
  - $T_I = \bigwedge_{s \in S \setminus F} (s'_R \rightarrow s'_L)$. This captures the requirement from Section 2.6.4 that the subset R' chosen must be a subset of L'.
  - $T_{LC} = \bigwedge_{s \in S} (s_L \rightarrow \rho(s)[S'_L/S])$; $\rho(s)[S'_L/S]$ refers to $\rho(s)$ with a substitution, where the variables in S are replaced with the corresponding variables in S'_L. This term expresses the requirement that the subsets of L' may be transitioned into from the current state.
  - $T_R = \left(F' \wedge \bigwedge_{s \in S \setminus F} (s'_L \rightarrow s'_R) \right) \vee \left(\neg F' \wedge \bigwedge_{s \in S \setminus F} (s_R \rightarrow \rho(s)[S'_{LR}/S]) \right)$; $\rho(s)[S'_{LR}/S]$ again refers to $\rho(s)$ with a substitution, though this time the variables in S are replaced with the corresponding variables in S'_R for the rejecting states, and the corresponding variables in S'_L for the accepting states. This captures the last clause of the construction outlined in Section 2.6.4.

  The overall transition relation is then $T_I \wedge T_{LC} \wedge T_R$. We construct the transition relation symbolically using BDDs; in particular, we found the `SwapVariables()` method of the CUDD package to be useful in setting up the substitutions here.

5.3.6 Structural Composition and Path Finding

MCMAS, by default, caches the transition relation from one time step to the next; we need to update this cache with the transition relation for our symbolic Büchi automaton. The `check_dynamic_automaton()` method updates this cache, and then performs CTL model checking for $EG\top$ with one additional fairness constraint (the aforementioned fairness constraint F').

Before we return the result of the check for $EG\top$, there are two postprocessing steps we need to carry out:

- We only want to consider the states in which the symbolic Büchi automaton is also in its initial state. This is obtained by taking the intersection of the result with a symbolic representation of said initial states. Recall from Section 2.6.4 that the starting state of the nondeterministic Büchi automaton is $(s^0, \emptyset)$ – we thus only want to consider the starting states where we have $\left(s_L^0 \wedge \bigwedge_{s \in S \setminus \{s^0\}} \neg s_L \wedge \bigwedge_{s \in S \setminus F} \neg s_R\right)$. This is implemented as follows:

```
BDD result_fair = check_EG_fair(para->bddmgr->bddOne(),
                                para, accepting_BDDs);

BDD variable_id = (state_map.find(initial_state))->second;
result_fair *= variable_id;
for (map<automaton_node*, BDD>::const_iterator it = state_map.begin();
     it != state_map.end(); ++it) {
  if (it->second != variable_id) {
    result_fair *= !(it->second);
  }
  if (rejecting_state_map.find(it->first) != rejecting_state_map.end()) {
    result_fair *= !(rejecting_state_map[it->first]);
  }
}
```

- Furthermore, we want to return the states *of the original model* where the property holds; in other words, any information about the states of the symbolic Büchi automaton itself should not be returned as a part of our result. This is implemented by selecting any states where, after the previous filtering step, there exists a valid starting configuration of the symbolic Büchi automaton, as follows:

```
for (map<automaton_node*, BDD>::iterator state_map_it = state_map.begin();
     state_map_it != state_map.end(); ++state_map_it) {
  result_fair = result_fair.ExistAbstract(state_map_it->second);
}
for (map<automaton_node*, BDD>::iterator
     rejecting_state_map_it = rejecting_state_map.begin();
     rejecting_state_map_it != rejecting_state_map.end();
     ++rejecting_state_map_it) {
  result_fair = result_fair.ExistAbstract(rejecting_state_map_it->second);
}
```

The `check_dynamic_automaton()` method concludes with a small amount of cleanup work.

Finally, we extend the `check_formula()` method of the `modal_formula` class to handle the modal cases for LDLK. Much like in our LTLK implementation, `check_formula()` should *never* be called directly on a diamond or box modality (`op` values 70 and 71 respectively); instead, the method should only be called on the LDL root (`op` value 80). We handle this case as follows:

```
1  modal_formula* nnf_formula
2    = (new modal_formula(3, (modal_formula*) obj[0]))->push_negations(0);
3
4  map<modal_formula*, BDD> epistemic_valuation;
5  nnf_formula->compute_epistemic_path_subformulae(para, &epistemic_valuation);
6  dynamic_automaton* da =
7    dynamic_automaton::construct_dynamic_automaton(nnf_formula, para,
8                                                   &epistemic_valuation);
9  result = !(da->check_dynamic_automaton(para));
```

We first convert the negation of the current formula to negation normal form (NNF) in line 2. We then compute the states in which all epistemic subformulae hold, using a similar approach to our implementations for LTLK and CTL*K as in Section 3.2.3 (line 5). We construct a dynamic automaton for our NNF formula, given the states in which all of the epistemic subformulae hold (line 7), and call its `check_dynamic_automaton()` method to find the states in which there exist an infinite path with the negation of our formula. Finally, we negate this to find the states in which all paths satisfy our formula (line 9).

5.3.7 Counterexample Generation

We implemented counterexample generation for LDLK formulae in a similar manner to that for LTLK and CTL*K. Suppose that the LDLK formula ϕ does not hold in all states; this means that there is some state, from which there exists some path along which ϕ does not hold. Thus, implementing LDLK counterexample generation involved constructing the symbolic Büchi automaton for $\neg\phi$, and then finding a witness for $EG\top$ in the composition of the model and the automaton. The `build_cex` method in `modal_formula` constructs a `dynamic_automaton` for $\neg\phi$ and then calls said `dynamic_automaton`'s own `build_cex` method; this, in turn, sets up the construction correctly and calls `build_cex` on $EG\top$. This is implemented as follows:

```
1  modal_formula* spec = new modal_formula(11, new modal_formula(5, NULL));
2
3  BDD result_fair = (state_map.find(initial_state))->second;
4  for (map<automaton_node*, BDD>::const_iterator it = state_map.begin();
5       it != state_map.end(); ++it) {
6    if (it->second != state_map.find(initial_state)->second) {
7      result_fair *= !(it->second);
8    }
9  }
10 *state *= result_fair;
11 bool successful_cex = spec->build_cex(state, index, para, countex, idbdd, cextr);
```

If MCMAS is invoked with the `-c` option, it will generate a suitable counterexample trace.

5.4 Performance Optimisations

We observed that our tool for model checking LDLK specifications could perform well on large state spaces, if specification formulae were very small; conversely, for large formulae, the tool did have some difficulty. Given that the LDLK model checking problem is PSPACE-complete, and our algorithm as discussed in Section 5.1.5 has runtime linear in the size of the model but exponential in the size of the formula, to some extent this may be unavoidable. Nonetheless, we did explore several possible avenues for improving the performance of our implementation.

We considered optimising our tool's behaviour in two dimensions – its performance on large *models*, as well as its performance on large *formulae*.

5.4.1 Analysis on Large Models

Consider the Prisoners scenario discussed in Section 8.7; we attempt to verify the formula $[\top^*]\neg Execute$ for $N = 20$ prisoners. The formula (and hence the automaton for the formula) is very small – however, the model has just over 64 billion states (MCMAS reports this as 6.41709×10^{10}). We observed using the `callgrind` profiler that the runtime (in terms of number of CPU cycles consumed) was dominated by computing the set of fair, reachable states. (Paths have been shortened and function arguments truncated for clarity.)

```
 20        reach = compute_reach(in_st, v, pv, a...);
 61  => ???:BDD::operator=(BDD const&) (1x)
49,028,820,105  => computereach.cc:compute_reach(...) (1x)
(...)
  2        do_model_checking(para);
295,498,144  => model_checking.cc:do_model_checking(bdd_parameters*) (1x)
```

The numbers on the left refer to the number of CPU cycles spent evaluating the relevant method. Observe that in this case, computing the set of fair reachable states required more than 165 times as many CPU cycles as actually performing model checking. This does suggest that it is not likely that our algorithm is very sensitive to increases in the size of the model, and also suggests that it may be more worthwhile to investigate optimising the large formula case instead of the large model case.

5.4.2 Analysis on Large Formulae

Again, we used the `callgrind` profiler to analyse the runtime behaviour of our implementation, and visualised it using the `kcachegrind` tool. Consider Figure 5.3, which shows profiling data for verifying the formula $\langle(\top;\top)^*\rangle(recack) \rightarrow \langle(\top)^*\rangle(recbit \wedge \neg recack)$[2] for the bit transmission protocol (introduced earlier in Section 2.2.2, with ISPL code in Appendix A). Note that in the bit transmission protocol, there are only 22 reachable states.

The profiling data suggests that the time-consuming step in this case involved carrying out the symbolic breakpoint construction (as opposed to, say, constructing the alternating automaton, or performing model checking in the composition of the tableau and the model; these would be reflected in `construct_dynamic_automaton()` or `check_EG_true()` respectively). We investigate this in greater detail using `callgrind_annotate`, which can give us an estimate of the number of CPU cycles spent executing each line of the source code. Consider the following excerpt from running `callgrind_annotate` on the output of `callgrind` (whitespace added and comments removed for readability):

```
  8    BDD symbolic_transition = t_I * t_LC * t_R;
78,129,202,979  => ???:BDD::operator*(BDD const&) const (2x)
  .
  2    agents->push_back(NULL);
  1    para->vRT->push_back(symbolic_transition);
  1    para->calReachRT = true;
  .
  .    vector<BDD> accepting_BDDs;
  .    accepting_BDDs.push_back(accepting_BDD);
  9    BDD result_fair = check_EG_fair(para->bddmgr->bddOne(), para,
                                       accepting_BDDs);
 48  => ???:Cudd::bddOne() const (1x)
```

[2]This property means "if the Sender receives an ack after some even number of states, then there is some point in the path where the Receiver had received the bit, but the Sender had not received the ack." This is clearly true, as the Receiver only begins sending acks once the bit has been received.

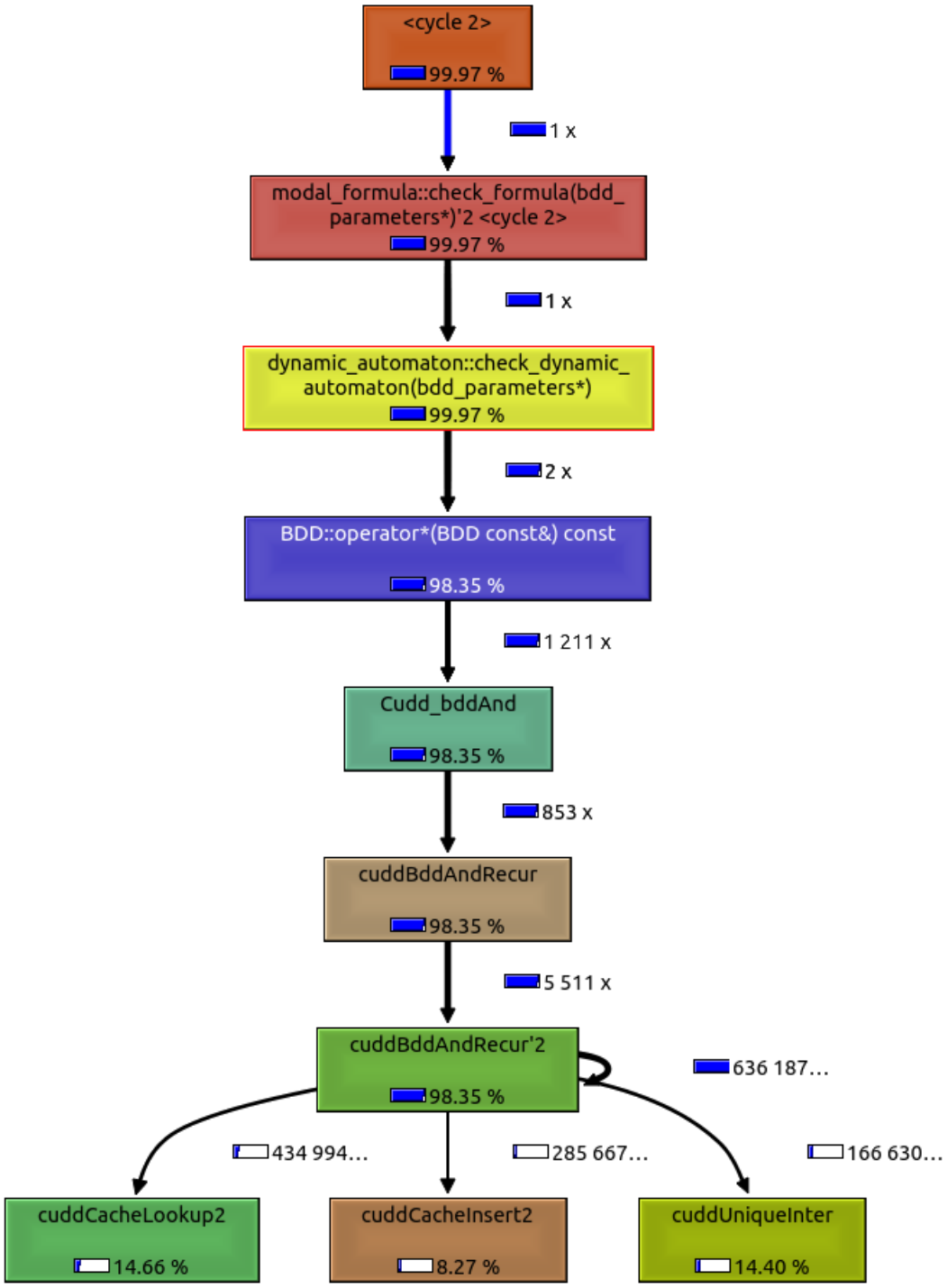


Figure 5.3: Approximation of CPU time used by each method when verifying $\langle(\top;\top)^*\rangle(recack) \rightarrow \langle(\top)^*\rangle(recbit \wedge \neg recack)$, visualised with `kcachegrind`. This highlights that `check_dynamic_automaton()` calls BDD `operator*` twice, but these two calls consumed 98.35% of the CPU cycles used by MCMAS.

```
1,271,713,062  => utilities.cc:check_EG_fair(BDD, bdd_parameters*,
  std::vector<BDD, std::allocator<BDD> >) (1x)
```

This confirms that the symbolic breakpoint construction is indeed where most of the time is being spent – the process of just composing t_I, t_{LC} and t_R itself already requires 61.4 times as many CPU cycles as the $EG\top$ model checking step – and we thus decided to focus our optimisation efforts toward this. We focused on improving performance in two ways:

1. Reduce the number of states in the alternating automaton. This would likely reduce the time required for said breakpoint construction. For each state of our alternating automaton, we require either 2 BDD variables (if it is accepting) or 4 (if it is not) in our symbolic Büchi automaton. Thus, each state that we are able to prune from the alternating automaton would reduce the Büchi automaton state space by a factor of 4 or 16 respectively.

2. Improve the efficiency of the way we compose t_I, t_{LC} and t_R (even though the final result we compute is the same). This involves exploiting several logical equivalences (in particular, the commutativity and associativity of $\wedge$) whilst considering the time complexity of BDD operations.

5.4.3 Automata Simplification

We introduced four measures to reduce the number of states in the alternating automaton, as follows.

1. **Minimise ϵ-NFAs to include only states that are directly reachable after non-ϵ transitions.** Consider the transition relations that we construct for our automata (Equations 5.2 and 5.3). Observe that if we have a state s that does not have any inbound non-ϵ transitions, the state will never actually be transitioned to in the alternating automaton (since it does not satisfy $s \in \rho_r(s', A)$ for any s' and A), and can thus be removed. We can thus construct the full marked ϵ-NFAs from the modal expressions, and filter the states to only consider those which have inbound non-ϵ transitions, before computing the transition relation and including them in our alternating automaton.

2. **Avoid constructing automata for non-temporal tests.** Consider that if we have a test for a *propositional* formula ϕ, whether the test succeeds or fails depends only on the current state. Thus, instead of constructing an automaton for this test and transitioning into it, it suffices to, for the relevant critical set, add a condition that we must be in a state where ϕ also holds. We check if the test formula is non-temporal using the `is_NonTemporal()` method of `modal_formula`. For example, we implement the construction of each term of the upper disjunct of Equation 5.2 for the $s \in S_r$ as follows:

```
if ((*prod_it)->is_NonTemporal()) {
  sum_term *= (*prod_it)->check_formula(para);
} else {
  modal_formula* nnf_formula = (*prod_it)->push_negations(0);
  dynamic_automaton* nnf_automaton
    = construct_dynamic_automaton(nnf_formula, para, epistemic_valuation);
  state_map.insert(nnf_automaton->state_map.begin(),
                   nnf_automaton->state_map.end());
  transition_mapping->insert(nnf_automaton->transition_mapping->begin(),
                             nnf_automaton->transition_mapping->end());
  sum_term *= nnf_automaton->transition_mapping
                ->find(nnf_automaton->initial_state)->second;
  accepting_states.insert(nnf_automaton->accepting_states.begin(),
```

```
14                                nnf_automaton->accepting_states.end());
15  }
16  required_tests += sum_term;
```

3. **Propositional shortcircuiting.** Suppose we have a *propositional* subformula ϕ (which possibly uses epistemic modalities). While constructing a suitable automaton representing this subformula is certainly correct, it is unnecessary as whether ϕ holds or not only depends on the current state. Thus, we can treat all such subformulae as new atomic propositions holding in precisely the states said propositional subformulae hold. This leads to a boost in performance, as we avoid constructing the full dynamic automaton representing ϕ; for example, if ϕ is the conjunction of several non-temporal properties, we can simply generate an automaton for ϕ that has just three states, rather than generating the alternating automaton for the conjunction, which introduces more than three states per conjunct (following Section 5.1.1).

4. **Conjunct partitioning.** Suppose we have a formula of the form $\phi \wedge \psi$. While we can construct an alternating automaton for both ϕ and ψ, it is likely to be substantially faster if we separately construct automata for ϕ and ψ, and independently use the symbolic breakpoint construction when building these automata (this gives us an $O(3^{|\phi|} + 3^{|\psi|})$-sized symbolic Büchi automaton construction, as opposed to $O(3^{|\phi|+|\psi|})$). This is safe:

   $\mathcal{I}, g \vDash \phi \wedge \psi$
   $\leftrightarrow$ on every path π starting at g we have $\mathcal{I}, \pi \vDash \phi \wedge \psi$
   $\leftrightarrow$ on every path π starting at g, both $\mathcal{I}, \pi \vDash \phi$ and $\mathcal{I}, \pi \vDash \psi$
   $\leftrightarrow$ (on every path π starting at g, $\mathcal{I}, \pi \vDash \phi$) and (on every path π starting at g, $\mathcal{I}, \pi \vDash \psi$)
   $\leftrightarrow \mathcal{I}, g \vDash \phi$ and $\mathcal{I}, g \vDash \psi$

   This is implemented by performing a check on the abstract syntax tree of the `modal_formula` before building the alternating automata, and rewriting our formulae accordingly.

5.4.4 Efficient Conjunct Computation

Recall that the symbolic breakpoint construction requires us to compute the conjunction of T_I, T_{LC} and T_R, which were defined in Section 5.3.5. The former two terms T_I and T_{LC} consist of many conjuncts, and owing to the commutative and associative properties of boolean AND, we can select an order to construct this conjunction that is efficient as far as BDD manipulation is concerned. Although T_R includes a disjunction, each of its disjuncts also consist of several conjuncts – thus, we can construct each disjunct in a way that is efficient for BDD manipulation.

This is significant, since from [23] we have that the complexity of an *apply* operation on two BDDs B_1 and B_2 is $O(|B_1| \times |B_2|)$. Choosing an efficient order of operations can have a considerable impact on runtimes. For example, consider the expression $\bot \wedge \phi \wedge \psi$, where ϕ and ψ have BDDs B_ϕ and B_ψ of size $N \gg 1$:

- Computing $(\bot \wedge (\phi \wedge \psi))$ requires us to evaluate $apply(\wedge, B_\phi, B_\psi)$, which has time complexity $O(N^2)$.
- Computing $((\bot \wedge \phi) \wedge \psi)$ requires us to evaluate $apply(\wedge, B_0, B_\phi)$ and $apply(\wedge, B_0, B_\psi)$; each of these operations has time complexity $O(N)$ and thus the overall time complexity is $O(N)$.

We first investigate $T_R = \left(F' \wedge \bigwedge_{s \in S \backslash F} (s'_L \rightarrow s'_R)\right) \vee \left(\neg F' \wedge \bigwedge_{s \in S \backslash F} (s_R \rightarrow \rho(s)[S'_{LR}/S])\right)$. We compute each disjunct using an accumulator approach, as we find this works well in practice. For instance, for the second disjunct, we have:

```
BDD rejection_term = (!accepting_BDD);
for (map<automaton_node*, BDD>::iterator afa_it = transition_mapping->begin();
     afa_it != transition_mapping->end(); ++afa_it) {
  if (accepting_states.find(afa_it->first) == accepting_states.end()) {
    BDD substitute_relation = afa_it->second.SwapVariables(q_L, q_LRp);
    rejection_term *= (!rejecting_state_map[afa_it->first] + substitute_relation);
  }
}
```

We did consider using the distributive property of $\wedge$ over $\vee$ to rewrite the expression as

$$T'_R = \left(\bigwedge_{\phi \in T_{RP}, \psi \in T_{RN}} (\phi \vee \psi)\right)$$

where $T_{RP} = \{F'\} \cup \{(s'_L \rightarrow s'_R) : s \in S \backslash F\}$, $T_{RN} = \{\neg F'\} \cup \{(s'_R \rightarrow \rho(S)[S'_{LR}/S]) : s \in S \backslash F\}$. However, this did not seem to be effective, owing to the quadratic blowup in the size of the formula. Even though each BDD in T_{RP} is likely to be very small, meaning that the cost of constructing the BDD for a single conjunct of T'_R is also likely to be small, this appeared (empirically) to be outweighed by the increased size of the formula. We thus compute T_R using an accumulator BDD for each conjunct, and then applying $\vee$ once (as outlined above).

Next, we investigate $T_I = \bigwedge_{s \in S \backslash F} (s'_R \rightarrow s'_L)$. Notice that by construction, the BDD representing each conjunct of T_I is very small, possibly with just 2 non-terminal nodes. However the BDD for T_I itself may potentially be very large, depending on the variable ordering used[3]. Furthermore, consider that T_R is likely to already have used many or all of the variables in S'_L or S'_R. We thus (generally speaking) do not anticipate a large increase in the BDD size, even as we compute the conjunction of T_R and the individual conjuncts of T_I. We found that using T_R as an accumulator and iteratively computing its conjunction with the conjuncts of T_I was substantially more efficient in practice than computing T_I first and then evaluating $apply(\wedge, T_I, T_R)$.

Our reasoning for $T_{LC} = \bigwedge_{s \in S} (s_L \rightarrow \rho(s)[S'_L/S])$ is similar (though it is not necessarily the case that each conjunct is as small as that in T_I, owing to the presence of the transition relation). We thus reuse our (accumulated) result of $T_R \wedge T_I$ as an accumulator, and iteratively compute its conjunction with the conjuncts of T_{LC}. This is implemented by constructing a vector of conjuncts and iterating through them:

```
for (map<automaton_node*, BDD>::iterator afa_it = transition_mapping->begin();
     afa_it != transition_mapping->end(); ++afa_it) {
  if (accepting_states.find(afa_it->first) == accepting_states.end()) {
    // q'R implies q'L
    target_BDDs.push_back((!next_rejecting_state_map[afa_it->first] +
                           next_state_map[afa_it->first]));
  }
}
for (map<automaton_node*, BDD>::iterator afa_it = transition_mapping->begin();
```

[3]That said, it may be linear as well, if we have a fortuitous variable ordering.

```
10      afa_it != transition_mapping->end(); ++afa_it) {
11    BDD substitute_relation = afa_it->second.SwapVariables(q_L, q_Lp);
12    target_BDDs.push_back(!state_map[afa_it->first] + substitute_relation);
13  }
14
15  // this is t_R, calculated above
16  BDD symbolic_transition = acceptance_term + rejection_term;
17  for (int i = 0; i < target_BDDs.size(); ++i) {
18    symbolic_transition *= target_BDDs[i];
19  }
```

Further to the above, we also attempted to compute the conjunction of T_R and the individual conjuncts of T_I and T_{LC} in parallel. We implemented this using OpenMP [9], a popular parallel programming framework:

```
1   #pragma omp parallel // Runs the following code block in parallel.
2   {
3     Cudd cudd(4 * transition_mapping->size(), 0, CUDD_UNIQUE_SLOTS,
4               options["cachesize"], options["maxmem"], 0);
5     Cudd_AutodynDisable(cudd.getManager());
6     Cudd_ShuffleHeap(cudd.getManager(), para->bddmgr->getManager()->invperm);
7     BDD privateBDD = cudd.bddOne();
8     #pragma omp for nowait // no need to synchronize at end of loop
9     for (int i = 0; i < target_BDDs.size(); ++i) {
10      privateBDD *= target_BDDs[i].Transfer(cudd);
11    }
12    #pragma omp critical // need to synchronize access, CUDD not thread-safe
13    {
14      symbolic_transition *= privateBDD.Transfer(*(para->bddmgr));
15    }
16  }
```

However, we found that this did not seem to improve performance, and in many cases even led to a slowdown. This is for several reasons – firstly, MCMAS' BDD package CUDD [3] is not thread-safe, and requires each thread to have a separate *BDD manager* which maintains hash tables and caches for BDDs. We thus need to manually set up a BDD manager for each thread (lines 3–6) and *transfer* each BDD from the original BDD manager to and from the thread's private manager (lines 10, 14). Furthermore, opportunities for caching to improve performance *across threads* are not exploited, because caches are implemented at the BDD manager level [3]. Secondly, as discussed earlier *apply* operations on BDDs scale with the product of the size of the BDD operands; hence, computing partial results and combining these can actually be significantly more expensive (and we found this to occur in practice).

We analyse in greater detail the impact of these performance optimisations in Chapter 8.

Chapter 6

Full Branching Time Dynamic Epistemic Logic (CDL*K)

In this chapter we formalise the full branching-time extension of LDLK, CDL*K; this allows us to specify properties concerning the *existence* of paths, which LDLK did not readily allow us to do. We show that CDL*K subsumes both LDLK and CTL*K. We then provide an algorithm for model checking CDL*K that builds on our LDLK algorithm, and show that it preserves the same asymptotic time complexity as our LDLK algorithm.

We then discuss our implementation of a tool for symbolic model checking of CDL*K specifications, along with how we handled counterexample generation and optimised its performance. The original work discussed in this chapter that was developed as part of this individual project is as follows:

1. **Formalism of the syntax and semantics of CDL*K.** To the best of our knowledge, this is a novel logic; no other work on CDL*K currently exists.

2. **CDL*K model checking algorithm.** We show that the construction of [39] is applicable to LDLK and CDL*K. We then outline how our LDL and LDLK algorithms from Chapter 5 can be used to check CDL*K properties.

3. **Proof that CDL*K is PSPACE-complete, with runtime exponential only in the size of the formula.** This means that CDL*K despite its high expressive power still maintains the same theoretical complexity as LTL.

4. **Extension of MCMAS to support CDL*K.** We concretely implemented our CDL*K model checking algorithm as an extension of MCMAS. This further augments the capabilities of MCMAS, as there are CDL*K properties that are neither expressible in CTL*K nor in LDLK (Lemma 6.3).

6.1 Syntax and Semantics

The syntax of CDL*K is defined as follows:

Definition 6.1. (CDL*K Path Formula) The syntax of a CDL*K *path formula* ψ is as follows:

$\psi ::= \phi$ where ϕ is a CDL*K state formula; this will be defined shortly.
$\quad | \quad \neg\psi$

| $\psi \wedge \psi$
| $\langle\rho\rangle\psi$

$\rho ::=$ ζ where ζ is a propositional formula
| $\psi?$
| $\rho + \rho$
| $\rho;\rho$
| ρ^*

We also include the abbreviations from propositional logic and for the box modality from LDL (that is, where ϕ and ψ are path formulas, $\bot = \neg\top$, $\phi \vee \psi = \neg(\neg\phi \wedge \neg\psi)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\phi \leftrightarrow \psi = (\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$ and $[\rho]\phi = \neg\langle\rho\rangle\neg\phi$).

Definition 6.2. (CDL*K State Formulas) The syntax of a CDL*K *state formula* ϕ is as follows:

$\phi ::=$ p where p is an atomic proposition
| $\top$
| $\neg\phi$
| $\phi \wedge \phi$
| $E\psi$ where ψ is a CDL*K path formula
| $K_i\phi$ where $i \in \Sigma$
| $E_\Gamma\phi$ where $\Gamma \subseteq \Sigma$
| $D_\Gamma\phi$ where $\Gamma \subseteq \Sigma$
| $C_\Gamma\phi$ where $\Gamma \subseteq \Sigma$

As before, we also include the abbreviations from propositional logic. (That is, where ϕ and ψ are state formulas, $\bot = \neg\top$, $\phi\vee\psi = \neg(\neg\phi\wedge\neg\psi)$, $\phi \rightarrow \psi = \neg\phi\vee\psi$ and $\phi \leftrightarrow \psi = (\phi \rightarrow \psi)\wedge(\psi \rightarrow \phi)$). We also add an additional abbreviation, $A\phi = \neg E\neg\phi$.

The semantics of CDL*K may be defined as follows:

Definition 6.3. (CDL*K semantics) Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E\}}, I, h\right\rangle$ be an interpreted system. Let $g \in G$ be a global state of the system and π be a path in $\mathcal{I}$. Then,

- $\mathcal{I}, g \vDash E\psi$ iff there exists a path π starting at g such that $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, g \vDash A\psi$ iff on all paths π starting at g, we have $\mathcal{I}, \pi \vDash \psi$.
- $\mathcal{I}, g \vDash \phi$ otherwise follows the semantics of CTLK.
- $\mathcal{I}, \pi \vDash \phi$ iff $\mathcal{I}, \pi(0) \vDash \phi$ where ϕ is a CDL*K state formula.
- $\mathcal{I}, \pi \vDash \psi$ otherwise follows the semantics of LDLK.

We observe that CDL*K is a relatively expressive specification language:

Lemma 6.1. Every LDLK property may be expressed in CDL*K.
Proof. Consider the LDLK property ϕ. We have $\mathcal{I}, g \vDash \phi$ if and only if on every path π starting at g we have $\mathcal{I}, \pi \vDash \phi$. However, this is clearly expressible as the CDL*K property $A\phi$, which has the same semantics.

Lemma 6.2. Every CTL*K property may be expressed in CDL*K.
Proof. We have the same constructs available for the state formulae, so it suffices to show that the CTL*K path formulae are expressible in CDL*K. We handle these by cases:

- $\mathcal{I}, \pi \vDash X\phi$ iff $\mathcal{I}, \pi^1 \vDash \phi$. This holds iff the suffix of π after dropping the first state satisfies ϕ, and thus holds iff $\mathcal{I}, \pi \vDash \langle\top\rangle\phi$.

- $\mathcal{I}, \pi \vDash \phi U \psi$ iff $\exists j \geq 0$ such that $\mathcal{I}, \pi^j \vDash \psi$ and for $0 \leq k < j$, $\mathcal{I}, \pi^k \vDash \phi$. This holds iff $\mathcal{I}, \pi \vDash \langle(\phi)^*\rangle\psi$, which holds iff there exists some prefix of π along which ϕ holds at every state, and ψ holds thereafter.

We revisit the ten statements we introduced in Section 2.3, and express them in CDL*K.

1. The train will leave the station in the next state: $A([\top](\text{train leaves}))$.

2. The train will not leave until the doors are closed: $A(\langle(\neg(\text{train leaves}))^*\rangle(\text{close doors}))$.

3. The train can leave at some point in the future: $E(\langle\top^*\rangle(\text{train leaves}))$

4. It is always possible for the train to leave the station: $A([\top^*]E(\langle\top^*\rangle(\text{train leaves})))$

5. The train will eventually come to a permanent stop: $A(\langle\top^*\rangle[\top^*](\text{stop}))$.

6. It is always possible for the train to leave the station **and** the train will eventually come to a permanent stop: $A(([\top^*]E(\langle\top^*\rangle(\text{train leaves}))) \wedge \langle\top^*\rangle[\top^*](\text{stop}))$.

7. The train always has the blue flag raised in the even states: $A([(\top;\top)^*](\text{blue flag raised}))$.

8. It is always the case that if train A is in the tunnel, train B knows the tunnel is occupied: $A([\top^*]((\text{A in tunnel}) \rightarrow K_B(\text{tunnel occupied})))$.

9. If train A knows that train B is in the tunnel, train B knows that train A knows, and train A knows that train B knows that train A knows, *et cetera*: $A([\top^*](K_A(\text{B in tunnel}) \rightarrow C_\Gamma(\text{B in tunnel})))$ where $\Gamma = \{A, B\}$.

10. If the train has the blue flag raised in the even states, then the controller eventually knows that the train can eventually come to a permanent stop:

$$A(([(\top;\top)^*](\text{blue flag raised})) \rightarrow \langle\top^*\rangle K_{Controller}(E(\langle\top^*\rangle[\top^*](\text{stop}))))$$

Thus, CDL*K is indeed considerably expressive; it subsumes all of the logics we have previously analysed in this project, as shown earlier with Lemmas 6.1 and 6.2, and is even more expressive.

Lemma 6.3. There exist properties expressible in CDL*K that are not expressible in either LDLK or CTL*K.

Proof. Property 10 in our example above is such a property. It mixes the universal and existential path quantifiers (which LDLK cannot express), and includes a statement concerning parity over a path (which CTL*K cannot express).

Nonetheless, it is worth noting that there certainly are temporal properties that are not expressible in CDL*K[1].

[1]For example, the property that "if p holds for N continuous states, then once p does not hold, q must immediately hold for N continuous states"; this corresponds to a non-regular language.

6.2 Algorithm

6.2.1 Recursive Descent

Our approach here is similar to that for extending our LTLK model checking tool to handle CTL*K model checking as discussed in Section 4.1.1. The algorithm for checking if an LDLK formula ϕ holds on all paths involves constructing a symbolic Büchi automaton for $\neg\phi$, composing it with the model, performing model checking of $EG\top$ with a suitable fairness constraint and finally complementing the result. This can easily be repurposed to check the CDL*K state formula $E\phi$, which involves constructing a symbolic Büchi automaton for ϕ, composing it with the model, and performing model checking of $EG\top$ with a suitable fairness constraint. We can thus directly adapt the LDLK algorithm to check CDL*K state formulae of the form $A\phi$ or $E\phi$, where ϕ is a CDL*K path formula; this can be applied to check arbitrary CDL*K formulae.

This is best illustrated with an example; consider the CDL*K formula

$$\phi = E(\langle((p? + q?); \top)^*\rangle r \wedge K_a(A([\top^*]p \rightarrow [\top^*]q)) \wedge E([(p; r)^*]q))$$

where p, q, r are atomic propositions. We proceed as follows.

1. First evaluate the states in which $A([\top^*]p \rightarrow [\top^*]q)$ (in red) holds. This has the same semantics as the LDLK formula $[\top^*]p \rightarrow [\top^*]q$, so we can find the states in which this holds.

2. Next, evaluate the states in which $K_a s$ (in blue) holds, where s is a fresh atomic proposition holding in precisely the states where $A([\top^*]p \rightarrow [\top^*]q)$ holds – we know this because we computed it in step 1. This can be computed using the epistemic accessibility relation for the agent a.

3. Next, evaluate the states in which $E([(p; r)^*]q)$ (in green) holds. This has the form $E\psi$ where ψ is a CDL*K path formula, so we can evaluate it.

4. Finally, evaluate the states in which $E(\langle((p? + q?); \top)^*\rangle r \wedge t \wedge u)$, where t is a fresh atomic proposition holding in precisely the states where $K_a s$ holds and u is another fresh atomic proposition holding in precisely the states where $E([(p; r)^*]q)$ holds. We know both of these, because we computed them in step 2 and step 3 respectively. Again, this has the form $E\psi$, where ψ is a CDL*K path formula, so we know how to evaluate this.

6.2.2 Complexity Analysis

Recall from Section 5.1.5 that the complexity of our LDLK algorithm is $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$. We claim that the algorithm above only requires $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time when combining the results for subformulae at each step; this would allow us to use Lemma 5.2 to conclude that the recursive descent procedure above preserves the $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time complexity. Observe that at each step in the recursion, we need to perform one or more of the following:

- Assign fresh atomic propositions to subformulae. This can be done in constant time for each subformula, assuming a suitable data structure, and hence requires $O(|\phi|)$ time in the worst case (since there can only be linearly many such subformulae).

- Find the states over which an epistemic modality holds. As discussed in Section 3.1.2 this can be done in $O(|\mathcal{I}|)$ time.

- Perform model checking of a formula $A\phi$ or $E\phi$, where ϕ is a CDL*K path formula. This can be done in $O(\max(2^{|\phi|\log|\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time, following our earlier result from Section 5.1.5.

- Evaluate a propositional formula over the states of the model (e.g. for a state formula $p \wedge q$). Observe that each operator can be evaluated in $O(|\mathcal{I}|)$ time since this involves a constant number of unions, intersections or complements over a set of states bounded by size $O(|\mathcal{I}|)$, and there are at most $O(|\phi|)$ such operators; so this would be bounded by $O(|\phi||\mathcal{I}|)$.

Hence, the overall additional "work" we may need to do requires $O(\max(2^{|\phi|\log|\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time. By Lemma 5.2 we have that our CDL*K model checking algorithm, as a whole, runs in $O(\max(2^{|\phi|\log|\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time. Notice that based on this result, the runtime is exponential only in the size of the formula (not in the size of the model).

6.3 PSPACE-Completeness of CDL*K

Theorem 6.1. Model checking of CDL*K is PSPACE-complete (assuming that the model $\mathcal{M}$ is given explicitly).

Proof. We first show PSPACE-hardness by reducing the problem of model checking LDL, which is known to be PSPACE-complete [96], to the CDL*K model checking problem. Suppose we have a model $\mathcal{M}$, a state s and a LDL formula ϕ. The LDL model checking problem then involves determining if $\mathcal{M}, s \vDash \phi$, which holds if on every path π starting from s, $\mathcal{M}, \pi \vDash \phi$. This matches precisely the semantics of the CDL*K formula $A\phi$ i.e.

$$\mathcal{M}, s \vDash_{\text{LDL}} \phi \leftrightarrow \mathcal{M}, s \vDash_{\text{CDL}^*} A\phi$$

where the turnstile subscripts indicate the logic involved. Thus, $(\mathcal{M}, s, \phi)$ is a yes-instance of LDL model checking if and only if $(\mathcal{M}, s, A\phi)$ is a yes-instance of CDL*K model checking. Constructing $A\phi$ from ϕ can clearly be carried out in polynomial time. Thus model checking of CDL*K is PSPACE-hard.

We then show PSPACE membership, by showing that the recursive descent approach presented in Section 6.2 operates in polynomial space if we use a polynomial space algorithm for LDL model checking (which exists, since LDL model checking is PSPACE-complete and hence in PSPACE). Consider that the maximum possible depth of the recursion is bounded by the length of the formula, and is thus polynomial in the input size (thus, it is safe for us to keep track of where we are in the recursion – for example, with a stack). Furthermore, at each step of the recursion, we may have to perform one or more of the following tasks, following our argument in Section 6.2.2:

- Assign fresh atomic propositions to subformulae. There can only be polynomially (linearly, in fact) many subformulae, and thus maintaining a mapping (say) of a fresh atomic proposition to a subformula can easily be done in polynomial space.

- Find the states over which an epistemic modality holds. The epistemic accessibility relation as well as the states over which said epistemic modality holds can be computed in polynomial space (a naïve algorithm could simply check each of the $O(n^2)$ pairs of states and record whether they are observationally equivalent to the relevant agents).

- Perform model checking of a formula $A\phi$, where ϕ is a path formula. We invoke the polynomial space LDL model checking algorithm on ϕ for this case.
- Perform model checking of a formula $E\phi$, where ϕ is a path formula. We invoke the polynomial space LDL model checking algorithm for checking $\neg\phi$, and take the complement of the result. This can be done in polynomial space; consider that if $f(x)$ is a polynomial, then $f(x+c)$ for constant c is also a polynomial, by the binomial expansion [33]. We can thus safely assert that the LDL model checking algorithm still runs in polynomial space; the complement can be done in space $O(|\mathcal{M}|)$, which is clearly polynomial in the input size.
- Evaluate a propositional formula over the states of the model (e.g. for a state formula $p \wedge q$). Evaluating an operator involves a constant number of unions, intersections or complements over sets of states bounded by size $O(|\mathcal{M}|)$. There are at most $O(|\phi|)$ such operators, and the result of each computation takes up $O(|\mathcal{M}|)$ space, so we would use at most $O(|\phi||\mathcal{M}|)$ space, which is certainly polynomial in the size of the input.

Thus, each of these steps can be carried out in polynomial space, and there is a polynomial number of such tasks (since the recursion depth is polynomial, and the number of tasks at each level of the recursion is at most constant; note that we have grouped similar tasks together). Hence, even if we store all of our intermediate results, the overall space used is still polynomial. Thus, CDL*K model checking can be done in polynomial space – i.e. it is in PSPACE.

Combining our results for hardness and membership gives us that CDL*K model checking is, indeed, PSPACE-complete.

It may be worth noting that our implementation for LDLK may not necessarily use polynomial space, because it constructs a symbolic Büchi automaton of exponential size using BDDs, which can use exponential space in the worst case [66]. Nonetheless, we believe this is outweighed by the average-case efficiency of BDDs.

6.4 Implementation

6.4.1 Overall Solution Architecture

We extended several existing objects and methods to add support for CDL*K model checking. We did not have to add any new classes for this; the collaborating objects and the roles they play are similar to that for LDLK (as described in Section 5.3.1). At a high level, we modify or extend MCMAS to perform the following:

- Parse CDL*K formulae (Section 6.4.2).
- Add support for model checking of $E\phi$, where ϕ is a CDL*K path formula (Section 6.4.3).
- Implement the recursive descent approach for dealing with nested path or epistemic modalities (Section 6.4.3).
- Generate counterexample paths for universally quantified CDL* formulae that do not hold, and generate witnesses for existentially quantified CDL*K formulae that do hold (Section 6.4.4).
- Extend the performance optimisations implemented for LDLK formulae to CDL*K formulae (Section 6.4.5).

6.4.2 Expression Parsing

We extended the implementation of the ISPL grammar to support parsing CDL*K formulae. The following changes were made to the grammar:

- As before, we prefer users to explicitly indicate their specification language of choice. A new terminal `CDLSPREFIX` which is "`CDL*`" was added, which allows users to specify a CDL*K formula.
- The `formlist` terminal was extended. This terminal parses lists of formulae; we extended it with several production rules allowing it to parse `cdlsformula`s.
- The `cdlsformula` nonterminal was added, which has production rules corresponding to the syntax of CDL*K state formulae.
- The `cdlspathformula` nonterminal was added, which has production rules corresponding to the syntax of CDL*K path formulae. For usability and clarity of output reasons (as discussed in Section 4.2.1), we opted to keep the terminals for LDL formulae and CDL*K path formulae separate.
- Finally, we added new versions of the `epistemicprefix` and `gepistemicprefix` nonterminals to support CDL*K formulae without respecifying that the formulae inside epistemic modalities would also be in CDL*K.

The syntax check step (`syntaxcheck.cc`) was also extended to verify that all atomic propositions in a CDL*K formula were well-defined.

6.4.3 Eϕ and Recursive Descent

Most of the complex logic for constructing a suitable alternating automaton, converting it into a symbolic Büchi automaton and composing it with the model was kept separate from the main `modal_formula` class. Previously, our method for an LDL formula ϕ involved converting $\neg\phi$ to negation normal form, building the automaton, checking $EG\top$ in the composition of the model and our automaton, and finally negating the result; this is the same approach we use for formulae of the form $A\phi$ (which have the same semantics).

The approach for $E\phi$ is very similar. However, the initial and final negations are unnecessary – we convert ϕ to negation normal form, build the automaton for it and check $EG\top$ in the composition of the model and our automaton. This is implemented as follows:

```
1  case 82: // CDL*'s E
2  {
3    modal_formula* nnf_formula = ((modal_formula*) obj[0])->push_negations(0);
4
5    map<modal_formula*, BDD> epistemic_valuation;
6    nnf_formula->compute_epistemic_path_subformulae(para, &epistemic_valuation);
7
8    dynamic_automaton* da =
9      dynamic_automaton::construct_dynamic_automaton(nnf_formula,
10       para, &epistemic_valuation);
11   result = da->check_dynamic_automaton(para);
12   break;
13 }
```

We also extend the `compute_epistemic_path_subformulae()` method to treat the CDL*K A and E tokens as top-level path subformulae. They will thus be computed first (line 6) and subsequently treated as fresh atomic propositions when we build the dynamic automaton. Similarly, we extended the `construct_dynamic_automaton()` method to be aware of the possibility of encountering these tokens, and to construct the automata accordingly, treating these as atomic propositions.

6.4.4 Counterexample and Witness Generation

The algorithm for counterexample or witness generation for CDL*K properties is very similar to that for LDLK; the paths found when we check $EG\top$ in the composition of the model and our automata are precisely the witness paths (for E-quantified formulae that are true) or counterexample paths (for A-quantified formulae that are false). We extended the `build_cex()` method of `modal_formula` to handle the additional cases.

6.4.5 Performance Optimisations

Since our CDL*K algorithm uses the same methods from `dynamic_automaton` as the LDLK algorithm, the optimisations for LDLK discussed in Section 5.4, apart from conjunct partitioning, are automatically applied to our CDL*K algorithm. We also added support for conjunct partitioning when dealing with CDL*K formulae of the form $A(\phi \wedge \psi)$. There was one further optimisation specific to CDL*K that we added – *disjunct partitioning*. This is the natural extension of conjunct partitioning (item 4 in Section 5.4) to E-quantified formulae; we rewrite formulae of the form $E(\phi \vee \psi)$ to $(E\phi) \vee (E\psi)$, as long as $\phi \vee \psi$ is not propositional (if it is, propositional shortcircuiting applies first):

$\mathcal{I}, g \vDash E(\phi \vee \psi)$

$\leftrightarrow$ there exists a path π starting at g we have $\mathcal{I}, \pi \vDash \phi \vee \psi$

$\leftrightarrow$ there exists a path π starting at g with $\mathcal{I}, \pi \vDash \phi$ or $\mathcal{I}, \pi \vDash \psi$

$\leftrightarrow$ (exists a path π starting at g with $\mathcal{I}, \pi \vDash \phi$) or (exists a path π starting at g with $\mathcal{I}, \pi \vDash \psi$)

$\leftrightarrow \mathcal{I}, g \vDash E\phi$ or $\mathcal{I}, g \vDash E\psi$

$\leftrightarrow \mathcal{I}, g \vDash E\phi \vee E\psi$

Thus as required the optimisation is safe.

Chapter 7

Finite Trace Semantics

The logics we have discussed in the previous chapters deal with infinite traces. However, the synthesis problem (that is, given a specification, generate a model that satisfies the specification) is very difficult in the infinite trace case, but often feasible in the finite trace case [35]. Furthermore, finite traces and synthesis over finite traces are sufficient for many real-world applications, such as planning in artificial intelligence [35] or monitoring business meta-constraints[1] [36].

In this chapter we first introduce LDL over finite traces (LDL_f), which was introduced by De Giacomo and Vardi in [35]. We then formalise the corresponding full branching time version with epistemic modalities – CDL*K with finite trace semantics ($\text{CDL*}_f\text{K}$). Next, we introduce an original algorithm for reducing $\text{CDL*}_f\text{K}$ model checking to CDL*K model checking, and show that $\text{CDL*}_f\text{K}$ is also PSPACE-complete. We conclude with discussing our extension of MCMAS to support verifying properties using these finite trace semantics. The original work discussed in this chapter that was developed as part of this individual project is as follows:

1. **Formalism of CDL*K over finite traces, $\text{CDL*}_f\text{K}$.** To the best of our knowledge, this interpretation of CDL*K is also novel.

2. **$\text{CDL*}_f\text{K}$ model checking algorithm, and proof of correctness.** We introduce the concept of a *path terminator* agent, and use this agent to reduce the $\text{CDL*}_f\text{K}$ model checking problem to the CDL*K model checking problem.

3. **Proof that $\text{CDL*}_f\text{K}$ is PSPACE-complete, with runtime exponential only in the size of the formula.** This shows that $\text{CDL*}_f\text{K}$ has the same theoretical complexity as LTL_f (LTL on finite traces), which is already PSPACE-complete [35].

4. **Extension of MCMAS to support finite trace semantics (LDL_fK and $\text{CDL*}_f\text{K}$).** We augment our CDL*K extension of MCMAS to support verifying LDLK and CDL*K properties over finite traces. To the best of our knowledge, this is the first model checking tool for these specification languages as well.

7.1 LDL over Finite Traces (LDL_f)

The syntax of LDL_f is precisely the same as that of LDL (as introduced in Section 2.3.4). However, the semantics are adjusted for the setting of finite traces:

[1] For example, for a payroll process, a constraint could be "employees cannot be paid severance pay until they leave the company".

Definition 7.1. (LDL$_f$ semantics) Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle$ be an interpreted system. Let π be a path in $\mathcal{I}$. Furthermore, let *Last* be a nonnegative integer, indicating the index of the last state of π that should be considered. Then,

- $\mathcal{I}, \pi, Last \vDash p$ iff $\pi(0) \in h(p)$.
- $\mathcal{I}, \pi, Last \vDash \top$.
- $\mathcal{I}, \pi, Last \vDash \neg\phi$ iff it is not the case that $\mathcal{I}, \pi, Last \vDash \phi$.
- $\mathcal{I}, \pi, Last \vDash \phi \wedge \psi$ iff $\mathcal{I}, \pi, Last \vDash \phi$ and $\mathcal{I}, \pi, Last \vDash \psi$.
- $\mathcal{I}, \pi, Last \vDash \langle \rho \rangle \phi$ iff there exists $0 \leq i \leq Last$ s.t. $(0, i) \in \mathcal{R}(\rho, \pi)$ and $\mathcal{I}, \pi^i, Last \vDash \psi$.

$\mathcal{R}$ is defined in the same way as in Definition 2.21.

Similar to LDL, for a global state $g \in G$, we have $\mathcal{I}, g \vDash \phi$ iff on every path π and every possible choice of *Last* starting at g we have $\mathcal{I}, \pi, Last \vDash \phi$.

7.2 CDL*K over Finite Traces (CDL*$_f$K)

Again, the syntax of CDL*$_f$K is identical to that of CDL*K; we adjust the semantics of the path formulae to match that of LDL$_f$ path formulae. In other words, as in LDL$_f$, we have $\mathcal{I}, \pi, Last \vDash \langle \rho \rangle \phi$ iff there exists $0 \leq i \leq Last$ such that $(0, i) \in \mathcal{R}(\rho, \pi)$ and $\mathcal{I}, \pi^i \vDash \psi$. More formally, we have

Definition 7.2. (CDL*$_f$K semantics) Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i \in \Sigma \cup \{E\}}, I, h \rangle$ be an interpreted system. Let $g \in G$ be a global state of the system and π be a path in $\mathcal{I}$. Then,

- $\mathcal{I}, g \vDash E\psi$ iff there exists a path π starting at g and value of *Last* such that $\mathcal{I}, \pi, Last \vDash \psi$.
- $\mathcal{I}, g \vDash A\psi$ iff on all paths π starting at g and all values of *Last*, we have $\mathcal{I}, \pi, Last \vDash \psi$.
- $\mathcal{I}, g \vDash \phi$ otherwise follows the semantics of CTLK.
- $\mathcal{I}, \pi, Last \vDash \phi$ iff $\mathcal{I}, \pi(0) \vDash \phi$ (where ϕ is a CDL*$_f$K state formula).
- $\mathcal{I}, \pi, Last \vDash \psi$ otherwise follows the semantics of LDL$_f$K (that is, LDL$_f$ extended with epistemic modalities).

We perform model checking of CDL*$_f$K by reducing the problem to that of model checking CDL*K. This is done by introducing an additional *path terminator* agent, whose purpose is simply to keep track of whether the current path is still "alive" or not.

Definition 7.3. (Path Terminator) A *path terminator* P is an agent in an interpreted system, which has local states $L_P = \{alive, dead\}$ and actions $Act_P = \{continue, stop\}$. Furthermore, $P_P(alive) = \{continue, stop\}$; $P_P(dead) = \{stop\}$. The evolution of P is such that P is *alive* after a *continue* action and *dead* after a *stop* action.

When performing verification, we want the agent to start in the "alive" state, indicating that the current path is still alive, but at some (nondeterministic) point transition to the "dead" state, indicating that the current path has ended. Furthermore, to ensure that we only consider *finite* traces, we add an additional fairness constraint, that the agent is dead. This removes the runs where the agent is alive infinitely often.

Now, observe that we can simulate any given value of *Last*, because it is possible for the path terminator to remain in the alive state precisely *Last* times, and then thereafter transition to the dead state. To enforce the changed semantics for $\langle\rho\rangle\,\phi$, we introduce a *finite translation* function, as follows:

Definition 7.4. (Finite Translation) Let ϕ be a CDL* path formula. Then, we define the finite translation of ϕ, $f(\phi)$, as follows:

- $f(p) = p$, where p is an atomic proposition
- $f(\neg\phi) = \neg f(\phi)$
- $f(\phi \wedge \psi) = f(\phi) \wedge f(\psi)$
- $f(\langle\rho\rangle\phi) = \langle\rho\rangle(\mathit{Alive} \wedge \phi)$, where *Alive* is an atomic proposition holding if and only if the path terminator is alive.

Theorem 7.1. Let $\mathcal{I} = \langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E\}}, I, h\rangle$ be an interpreted system; furthermore, let $\mathcal{I}' = \langle (L_i, Act_i, P_i, t_i)_{i\in\Sigma\cup\{E,PathTerminator\}}, I', h'\rangle$ be an interpreted system, where I' is I with *Alive* true, and h' has the *Alive* atomic proposition. Then, for a given path π in $\mathcal{I}$ and nonnegative integer *Last*, let π' be a path in $\mathcal{I}'$, that is π augmented with the path terminator being alive in the first *Last* steps and dead thereafter. Let $\mathcal{I}, \pi \vDash_{\text{FINITE}} \phi$ denote ϕ holding on π over finite traces, and $\mathcal{I}, \pi \vDash_{\text{ININITE}} \phi$ denote ϕ holding on π over infinite traces. Then,

$$\mathcal{I}, \pi, \mathit{Last} \vDash_{\text{FINITE}} \phi \leftrightarrow \mathcal{I}', \pi' \vDash_{\text{INFINITE}} f(\phi)$$

Proof. Consider that the semantics are identical apart from the $\langle\rho\rangle\phi$ case. We handle this case by structural induction; we assume as the inductive hypothesis that $\mathcal{I}, \pi, \mathit{Last} \vDash_{\text{FINITE}} \phi \leftrightarrow \mathcal{I}', \pi' \vDash_{\text{INFINITE}} f(\phi)$, and also that $\mathcal{R}(\rho, \pi) = \mathcal{R}(\rho, \pi')$. Throughout the rest of the proof, we define $\vDash$ unqualified as $\vDash_{\text{INFINITE}}$. Observe that we have

$$\begin{aligned}
&\mathcal{I}', \pi' \vDash_{\text{INFINITE}} f(\langle\rho\rangle\phi) \\
\leftrightarrow\ &\mathcal{I}', \pi' \vDash \langle\rho\rangle(\mathit{Alive} \wedge f(\phi)) \\
\leftrightarrow\ &\exists i \geq 0 \text{ s.t. } (0,i) \in \mathcal{R}(\rho, \pi') \text{ and } \mathcal{I}, \pi'^i \vDash \mathit{Alive} \wedge f(\phi) \\
\leftrightarrow\ &\exists i \geq 0 \text{ s.t. } (0,i) \in \mathcal{R}(\rho, \pi') \text{ and } \mathcal{I}, \pi'^i \vDash \mathit{Alive} \text{ and } \mathcal{I}, \pi'^i \vDash f(\phi) && \text{by definition of } \wedge \\
\leftrightarrow\ &\exists i \geq 0 \text{ s.t. } (0,i) \in \mathcal{R}(\rho, \pi') \text{ and } (i \leq \mathit{Last}) \text{ and } \mathcal{I}, \pi'^i \vDash f(\phi) && \text{by construction of } \pi' \\
\leftrightarrow\ &\exists 0 \leq i \leq \mathit{Last} \text{ s.t. } (0,i) \in \mathcal{R}(\rho, \pi') \text{ and } \mathcal{I}, \pi'^i \vDash f(\phi) \\
\leftrightarrow\ &\exists 0 \leq i \leq \mathit{Last} \text{ s.t. } (0,i) \in \mathcal{R}(\rho, \pi) \text{ and } \mathcal{I}, \pi^i, \mathit{Last} \vDash_{\text{FINITE}} \phi && \text{by the inductive hypotheses} \\
\leftrightarrow\ &\mathcal{I}, \pi, \mathit{Last} \vDash_{\text{FINITE}} \phi
\end{aligned}$$

as required.

Since the above theorem applies for all values of *Last*, and we can simulate all possible values of *Last* by giving the path terminator nondeterministic behaviour as described above, we have reduced CDL*$_f$K model checking to CDL*K model checking, for which we can use our existing tool.

We conclude this section with a proof concerning the complexity of this logic.

Theorem 7.2. CDL*$_f$K model checking is PSPACE-complete (assuming that the model $\mathcal{M}$ is given explicitly).

Proof. We first show PSPACE-hardness, by reduction from LDL_f model checking, which was shown to be PSPACE-complete in [35]. Suppose we have a model $\mathcal{M}$, a state s and a LDL_f formula ϕ. The LDL_f model checking problem then involves determining if $\mathcal{M}, s \vDash \phi$, which holds if on every path π starting from s and every value of $Last$, $\mathcal{M}, \pi, Last \vDash \phi$. This matches the semantics of the $\text{CDL}^*{}_f\text{K}$ formula $A\phi$ i.e.

$$\mathcal{M}, s \vDash_{\text{LDL}_f} \phi \leftrightarrow \mathcal{M}, s \vDash_{\text{CDL}^*{}_f\text{K}} A\phi$$

where the turnstile subscripts indicate the logic involved. Thus, $(\mathcal{M}, s, \phi)$ is a yes-instance of LDL_f model checking if and only if $(\mathcal{M}, s, A\phi)$ is a yes-instance of $\text{CDL}^*{}_f\text{K}$ model checking. Constructing $A\phi$ from ϕ can clearly be carried out in polynomial time. Thus model checking of $\text{CDL}^*{}_f\text{K}$ is PSPACE-hard.

Membership in PSPACE follows by the reduction presented above. Since as demonstrated above we can reduce $\text{CDL}^*{}_f\text{K}$ model checking to CDL*K model checking, which as shown in Section 6.3 is in PSPACE, and PSPACE is closed under reduction [97], membership follows. Note that the reduction can easily be done in polynomial time, since we add 1 agent which has 2 states (so, at most, we need to generate linearly many more additional states), add 1 (constant) fairness constraint and compute the finite translation (which is linear in the size of the formula). Hence, as required, $\text{CDL}^*{}_f\text{K}$ model checking is PSPACE-complete. The runtime is clearly also polynomial in the size of the model, since the reduction is doable in time polynomial in the size of the model, it increases the model size only by a constant factor, and the CDL*K model checking step is also, itself, polynomial in the size of the model. Thus, we have exponentiality only in the size of the formula.

7.3 Implementation

We introduced or extended several files from MCMAS to add support for $\text{CDL}^*{}_f\text{K}$ model checking:

- `ldlf.cc` which contains a collection of utility methods for constructing the path terminator, computing the finite translation function over modal formulae, as well as string constants relating to these;
- `ldlf.hh` which contains an interface to the relevant utility methods and string constants;
- `read_options.cc` which is updated to recognise the new flag we introduce (Section 7.3.1);
- `modal_formula.cc` which is updated to invoke the finite translation function (from `ldlf.cc`) on LDL_fK and $\text{CDL}^*{}_f\text{K}$ formulae if the flag is set;
- `main.cc` which is updated to call into the relevant methods from `ldlf.cc` to set up the path terminator agent, if the flag is set.

We also added several tests over finite trace semantics, and extended the `check.py` script to call MCMAS with the correct command-line options for these tests.

7.3.1 Specifying Finite Trace Semantics

We added a new command-line flag `-ldlf` that allows users to specify that they want LDLK and CDL*K properties to be verified over finite traces. This was chosen as opposed to having the user specify this on a per-formula basis (e.g. using "`LDLf <p*>[q*]r`" to mean the LDL

formula $\langle p^* \rangle [q^*] r$ verified over finite traces), primarily because MCMAS constructs the full model before verifying any properties on it, and the path terminator agent that we introduce for verifying LDL$_f$K or CDL$^*{}_f$K would cause the size of our model to double. This could slow down verification of the properties specified over infinite traces (since all of our model checking algorithms are linear in the size of the model). Note that the inclusion of this flag does *not* affect the results returned by the algorithms for the other logics; for instance, an LTL formula will always be verified with infinite trace semantics even if the `-ldlf` flag is passed.

7.3.2 Input File Preprocessing

Observe that the path terminator agent can be expressed in ISPL as follows:

```
1  Agent PathTerminator
2    Vars:
3      alive : boolean;
4    end Vars
5    Actions = {continue, stop};
6    Protocol:
7      alive=true : {continue, stop};
8      alive=false: {stop};
9    end Protocol
10   Evolution:
11     alive=true if (Action=continue);
12     alive=false if (Action=stop);
13   end Evolution
14 end Agent
```

We did initially consider simulating the creation of such an agent internally. However, the logic for creating such an agent was somewhat convoluted; it seemed distributed over various parts of the parsing and syntax checking stages. We thus found it considerably easier to, instead, construct a temporary file with the additional agent, evaulation variables, specific initial states and fairness formulae. (The latter three requirements could have been established internally without too much difficulty; however, given that we were constructing such a temporary file, we decided that it was very simple to add these constructions here.)

We implement the creation of such a temporary file in the `ldlf.cc` class; we have a method `void build_ldlf_file(string filename)` that, given a string filename (which is typically taken from the command line arguments supplied to MCMAS), creates a temporary file with the additional constructions. This method relies on patterns in the syntax of an ISPL file (see definition 2.45), as well as an internal `guarded_find` method that helps us find the locations of critical tokens. We add the aforementioned constructs in the following positions:

- We add the `PathTerminator` agent just before the Evaluation section (that would be at the end of the Agents section). In practice, we actually add `PathTerminatorN`, where N is the smallest natural number for which this string is never found in the original ISPL file. This is done to avoid potential naming conflicts.

- We add one propositional variable to the `Evaluation` section, *Alive* holding if and only if `PathTerminatorN.alive = true`; N was the number from the previous step. In practice, again to avoid conflicts with other evaluation variables, we in practice add `AliveM`, where M is the smallest natural number for which this string is not found in the ISPL file.

- We add to the `InitStates` the constraint that the path terminator is initially alive (i.e. `PathTerminatorN.alive=true`). This is added as a conjunct to the existing `InitStates`.

- It is uncertain what the semantics of `Fairness` constraints are meant to be along a finite path; the LDL/CDL* interpretation of a fairness constraint p, $[\top^*]\langle\top^*\rangle p$ over finite trace semantics holds iff p holds in the last state being considered on a path. Currently, we ignore the user's `Fairness` section and replace it with our own, with `!AliveM` as the only fairness constraint (or add a `Fairness` section if none exists since this is optional in ISPL). For usability reasons, we print a warning if the input file did contain a `Fairness` section, highlighting that any fairness constraints the user may have specified are being ignored.

We then use this temporary file as input to the parser, which suitably constructs the relevant agents, propositions *etc.* following the "existing" workflows of MCMAS.

7.3.3 Finite Translation Function

We have notably *not* included the finite translation function in the previous step, owing to technical reasons – this translation is inductively defined, and we find that it is much more easily done *after* the parser has done its work and generated a formation tree for the formula.

We implement this translation function as `ldlf_augment()` in `ldlf.cc`; it consumes a pointer to a modal formula (in practice, a *path* formula) and returns a pointer to a new formula corresponding to the finite translation of the input. The method is implemented following said translation – for the diamond modality, this is done as follows:

```
1  case 70: // diamond
2  {
3    string* eval_property_name = new string(Ldlf::EVAL_NAME);
4    modal_formula* conjunct_augmented
5      = new modal_formula(1,
6          ldlf_augment((modal_formula*) (formula->get_operand(1))),
7          new modal_formula(new atomic_proposition(eval_property_name)));
8    return new modal_formula(70,
9      (ldl_modality*) (formula->get_operand(0)),
10     conjunct_augmented);
11 }
```

For the diamond case we have implemented the finite translation from Definition 7.4. For the box case we have implemented $f([\rho]\phi) = [\rho](Alive \to f(\phi))$; consider that this is correct, because we have

$$\begin{aligned}
f([\rho]\phi) &= f(\neg\langle\rho\rangle\neg\phi) && \text{since } [\rho]\phi = \neg\langle\rho\rangle\neg\phi \\
&= \neg f(\langle\rho\rangle\neg\phi) && \text{by definition of } f \\
&= \neg\langle\rho\rangle(Alive \wedge f(\neg\phi)) && \text{by definition of } f \\
&= \neg\langle\rho\rangle(Alive \wedge \neg f(\phi)) && \text{by definition of } f \\
&= \neg\langle\rho\rangle\neg(\neg Alive \vee f(\phi)) && \text{De Morgan's laws, double negation} \\
&= \neg\langle\rho\rangle\neg(Alive \to f(\phi)) && \text{definition of implies} \\
&= [\rho](Alive \to f(\phi)) && \text{since } [\rho]\phi = \neg\langle\rho\rangle\neg\phi
\end{aligned}$$

This function is called from the `check_formula()` method of `modal_formula` prior to model checking a formula of the form $A\phi$, $E\phi$ or an LDLK formula ϕ, if the `-ldlf` flag is set. It is worth noting that this check takes place *after* we check whether propositional shortcircuiting, discussed in Section 5.4, is applicable; this is safe since from Definition 7.4 propositional formulae will not change under finite translation. Note that because we convert the CDL*$_f$K model checking problem to a CDL*K model checking problem, the optimisations we have implemented in Sections 5.4 and 6.4.5 are all still applicable.

Chapter 8

Experimental Evaluation

In this chapter we first outline how to install and use MCMAS-Dynamic, the extension of MCMAS that we have developed to support Ltlk, Ctl*k, Ldlk and Cdl*k specifications (as well as finite trace semantics for the latter two). We then discuss the set of acceptance tests we have constructed, to give us evidence that our tool behaves as expected and to allow us to make changes more confidently.

We then discuss our setup for investigating the performance of our tool. Finally, we analyse our tool's performance over several scalable scenarios, considering how it scales with model and specification size, as well as how the algorithms for the varying specification languages compare.

8.1 Installation and Usage

MCMAS is distributed as an open source tool; the latest version may be downloaded from [10]. The user can download a zipped archive of the source files, and then run `make` to build the CUDD BDD package [88] and MCMAS.

The installation and build process for MCMAS-Dynamic is similar. It is worth noting, however, that the installation process for both tools is actually dependent on several additional utilities which are required to build the program:

- `make` to run suitable commands to build the tool,
- `flex` and `bison`, which are used to generate a parser from the MCMAS grammar files,
- a suitable C++ compiler (one can set this in the `Makefile`; the default is `g++`), and
- *(Optional, and newly added)* `python` for the various testing utilities (sanity checks, as well as performance benchmarking).

These utilities, if not already available, can be installed using `apt-get`. After `make` has completed, it may be useful to first run the `check.py` script in the `test` directory, which runs a suite of system tests to uncover possible problems (whether with the build process or otherwise).

MCMAS-Dynamic is generally used in a similar way to MCMAS; the usage of MCMAS itself is described in Section 2.8.1. It is invoked from the command line, accepts suitable arguments, and processes the input ISPL file (at a high level, following the steps outlined in Section 2.8.2). Our

extension does support some of the existing command line options (in particular, counterexample generation with `-c` and verbose output with `-v`). Furthermore, we added one command line option; if one wants to verify LDLK or CDL*K properties over finite traces, one should specify the `-ldlf` flag. MCMAS-Dynamic then produces output:

```
1  ****************************************************************************
2                    MCMAS-Dynamic v1.2.3
3
4   This software comes with ABSOLUTELY NO WARRANTY, to the extent
5   permitted by applicable law.
6
7   Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
8   Please send any feedback to <mcmas@imperial.ac.uk>
9  ****************************************************************************
10
11 Command line: ./mcmas -ldlf examples/strongly_connected.ispl
12
13 Warning: LDL/CDL* on finite traces is incompatible with fairness constraints.
14 (it's not possible to have something infinitely often on a finite path.)
15 examples/strongly_connected.ispl has been parsed successfully.
16 Global syntax checking...
17 Done
18 Encoding BDD parameters...
19 Building partial transition relation...
20 Building BDD for initial states...
21 Building reachable state space...
22 Checking formulae...
23 Building set of fair states...
24 Verifying properties...
25   Formula number 1: (F (G stop)), is TRUE in the model
26   Formula number 2: <(tt)*>[(tt)*]stop, is FALSE in the model
27   Formula number 3: (A <(tt)*>stop), is TRUE in the model
28 done, 3 formulae successfully read and checked
29 execution time in ms = 25 (0.025 s)
30 number of reachable states = 12
31 BDD memory in use = 9832784
```

There are a few differences with "vanilla" MCMAS:

- Of course, we see here the verification of LTL, LDL and CDL* properties (lines 25–27, respectively), which MCMAS did not previously support.
- We observe the new flag `-ldlf` for *finite trace semantics* (line 11), which the LDL and CDL* properties were verified over. We also observe the warning that the user's fairness constraints were overridden (lines 13–14).
- The model under examination is that from Figure 2.2 with s_1 as the only starting state. We note that the LTL property is still verified over infinite traces (line 25; it remains true, while the LDL property in line 26 which has the same semantics over infinite traces is actually false when interpreted over finite traces – consider the finite trace s_1, s_2).
- We also observe that the execution time output is now given more precisely (line 29). The reason for this change is discussed in Section 8.3.

8.2 Acceptance Tests

We paid significant attention to testing throughout the development process; this was important in giving us evidence that our implementation was sound and indeed behaved correctly.

Furthermore, as many of our procedures for model checking depended on all cases being handled correctly (for example, the alternating automaton construction for LDL formulae), having a robust testsuite would allow us to more quickly diagnose and fix bugs. Maintaining such a testsuite would also help us make changes more confidently, as we could easily gain at least some insight as to whether we had accidentally broken a different part of our program. This would have been tricky with the base version of MCMAS, which does not come with any form of testing (unit, integration or otherwise).

8.2.1 System Tests

Unfortunately, the existing design of MCMAS is not particularly amenable to unit testing. This is because unit testing involves us testing the behaviour of objects or methods in isolation; the existing design of MCMAS does have some issues which inhibited unit testing, as follows.

- MCMAS has several global variables (declared in `utilities.hh`) which are first populated whilst building the model. Many key utility methods (such as computing preimages) expect these variables to already be populated correctly; furthermore, there are certain undocumented invariants that are assumed. This makes testing said methods in isolation difficult, as one has to ensure that the relevant global variables are populated correctly.
- There are also several large, complex objects with many different responsibilities. The most egregious offender here is the `bdd_parameters` struct, which controls usage and caching of BDDs as well as several other parameters. Suitably populating these objects as well as ensuring that all invariants that hold in an actual run of the program also hold in test runs would be a fairly difficult and time-consuming undertaking.

A possible solution to the above could be to invest some time and effort in refactoring and adding tests to the existing MCMAS codebase. However, the time investment required would be substantial, and we find that this is also not the focus of our project.

We thus decided to implement tests at a coarser granularity that would still allow us to verify the behaviour of our program. This involved adding system-level tests, in the form of ISPL files that specify a model, formulae and expected answers (true or false) for each formula. We could then invoke MCMAS on said test file, and compare the actual output with what was expected. This allowed us to achieve some test coverage without requiring us to refactor any of the underlying MCMAS code.

To automate the above process, we wrote a Python script `check.py` that automatically runs tests on several ISPL test files, parsing them as well as interpreting MCMAS's output to determine if the actual results MCMAS produces matches our pre-specified expected result.

Our automated testing proved especially useful when implementing support for CTL*K; when we refactored parts of the grammar as well as the LTLK model checking logic (treating LTLK formulae as A-quantified CTL*K formulae to avoid code duplication), we found that we could quickly and easily check if we had accidentally broken the LTLK model-checking logic. We faced similar issues (and again, found our testsuite very useful) when extending our LDLK model checker to support CDL*K.

Our testsuite covers a fairly wide variety of specifications across the four languages for which we have implemented model checking, including complex formulae with substantial nesting of modalities (for LDLK and CDL*K). We also added tests for LDLK and CDL*K over finite traces.

We found this very useful in giving us confidence that our changes at each step (especially performance optimisations) did not affect the correctness of our implementation.

Unfortunately, we did not add tests to check the correctness of counterexample generation. The main problem we encountered was that the algorithm for generating witnesses for $EG\top$ in MCMAS is *nondeterministic*; it relies on the CUDD library's `PickOneMinterm()` method which randomly selects a state from which the formula holds [3]. Furthermore, returning minimal counterexamples in general is NP-complete [29], and we did not find an easy way of obtaining a canonical form for counterexamples. Verifying that a counterexample returned was correct in general would involve writing a verifier that could parse MCMAS' witness output and check that the output was indeed a valid counterexample, which seems excessively complex (and is not the focus of our project). This problem could perhaps be mitigated by developing models where there only exists one witness or counterexample path, though we did not pursue this in the interest of time.

8.2.2 Differential Testing

I adapted the idea of *differential testing* which is frequently used in compiler testing, where the same input is given to several programs purporting to carry out the same function, and mismatches in output could indicate bugs [37]. In the context of MCMAS, this relies on the fact that there are many properties that can be expressed in multiple specification languages; a mismatch in our answers from the various algorithms indicates a bug (as there is only one correct answer). I wrote several specifications for the bit transmission protocol that could be expressed in multiple specification languages and ensured that MCMAS returned the same result for all of the specifications. An example is as follows (more examples may be found in Appendix A):

```
1  Formulae
2    -- The bit is always eventually received.
3    -- False, since the environment may not work.
4    AF recbit;
5    LTL F recbit;
6    CTL* A(F recbit);
7    LDL <tt*> recbit;
8    CDL* A(<tt*> recbit);
9
10   -- Always possible to eventually receive the bit.
11   -- True (there's no amnesia or other reason why it becomes impossible)
12   AG EF recbit;
13   CTL* A(G(E(F(recbit))));
14   CDL* A([tt*](E(<tt*>(recbit))));
15 end Formulae
```

While useful, the strategy is limited to some extent since it can only expose bugs that occur when dealing with formulas in the common fragment of the specification languages under consideration. For example, LDLK and CTL*K have incomparable expressive power (as shown in Section 2.3.4); this technique will not be able to find bugs that only manifest when dealing with LDLK formulae that are inexpressible in CTL*K, and vice versa. Furthermore, even if a discrepancy is found, it may not immediately be clear *which* answer is wrong (though the cases we have designed are sufficiently simple that we are able to manually determine the right answer).

Another limitation of this approach is that each of the logics we implemented *added* expressive power, in the sense that there exist specifications in each language that could not be expressed in the previous languages implemented up to that point – for example, there exist properties

expressible in CDL*K that cannot be expressed in CTLK, LTLK, CTL*K or LDLK. Thus, this technique will not be able to provide complete coverage of each specification language. Furthermore, interpreting the same formula over finite and infinite trace semantics can lead to vastly different results; properties such as $[\top^*]\langle\top^*\rangle\phi$ ("ϕ occurs infinitely often" in LDLK) can change completely (to "ϕ holds on the last step of the path" in LDL_fK).

8.3 Performance Test Setup

We ran MCMAS-Dynamic on virtual machines in the Department of Computing, using the department's Apache CloudStack [1] service. The machines we used for testing ran Ubuntu 14.04 LTS, had two 2.7 GHz CPU cores and 16 GB of RAM. With a suitable verbosity level (`-v 4`), MCMAS will print output which indicates how much time was spent on each step of the encoding (such as encoding the ISPL model with BDD variables, computing the set of fair and reachable states, and model checking *each* formula). We determined that the impact of these print statements on the runtime is unlikely to be significant, since we only add a small number of print statements – for example, we add 1 print statement for each step in computing the depth of the model, as well as only 1 print statement per formula being checked. We did, however, have to fix an issue with the way MCMAS presents its time output, which could lose precision:

```
cout << "execution time = "
     << ((tmb1.time-tmb.time) + (tmb1.millitm-tmb.millitm)/1000.0) << endl;
```

The above calculation with floating point numbers has limited precision; on the lab machines we used, this only seemed to give us about 7 decimal digits of precision – for example, printing $100,000.0+0.001$ returned simply 100000. One possibility could be to use `long double`s though we decided to simply return the (integer) number of *milliseconds* used for the purpose of these tests. We maintain the display of the approximate number of seconds as well, for readability.

```
cout << "execution time in ms = "
     << (tmb1.time-tmb.time)*1000 + (tmb1.millitm-tmb.millitm)
     << " ("
     << ((tmb1.time-tmb.time) + (tmb1.millitm-tmb.millitm)/1000.0)
     << " s)" << endl;
```

To mitigate the effects of random noise on runtimes, we also verify each specification three times and take the median of the results. We also enforce a timelimit of 10^5 seconds for any given run.

We considered five different scenarios in our evaluation:

1. **Dining Cryptographers** (Section 8.4); this model allows us to test how our algorithms for LTLK and LDLK scale against the "basic" MCMAS algorithm for CTLK with models and formulae of increasing size. This scenario also showcases the effectiveness of our LDLK optimisations.

2. **Counter** (Section 8.5); this model is particularly amenable to LDL and CDL* properties concerning parity that are not expressible in CTL*.

3. **Bit Transmission** (Section 8.6); our analysis on this model focuses on testing how our algorithms scale with a model of constant size, but formulae of increasing size.

4. **Prisoners and a Lightbulb** (Section 8.7); this model allows us to test how our algorithms scale with formulae of constant size, but a rapidly increasing model. We also consider more complex properties that are only expressible in LDLK or CDL*K.

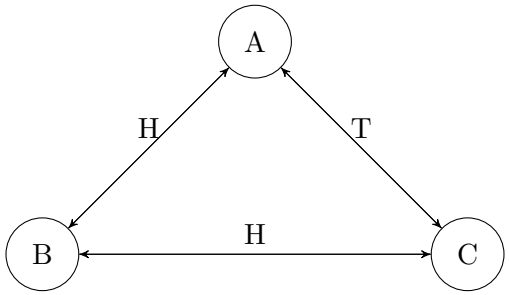


Figure 8.1: Possible instance of the Dining Cryptographers protocol. The arrows are labelled with the results of the coin flips that the relevant cryptograhers see.

5. **Go-Back-N** (Section 8.8); we model this automatic repeat-request network protocol in ISPL as a practical example of some of the capabilities of CDL*K, and evaluate various properties about this protocol over both infinite and finite traces.

8.4 Dining Cryptographers

The *dining cryptographers* protocol, proposed by Chaum in [27] involves multi-party computation[1] of the boolean OR function with the restriction that at most one bit is 1. The problem may be formulated as follows, quoting from [27]:

> *Three cryptographers are sitting down to dinner at their favorite three-star restaurant. Their waiter informs them that arrangements have been made with the maître d'hôtel for the bill to be paid anonymously. One of the cryptographers might be paying for the dinner, or it might have been the NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if the NSA is paying.*

The protocol works as follows (note that this protocol is clearly extensible to $N > 3$ cryptographers, following the same logic):

1. Each cryptographer flips a fair coin in secret.
2. They share this result with the cryptographer on their right. (We assume the cryptographers are seated in a circle.)
3. Each cryptographer announces if the two coins he can see are the same or different. If a cryptographer paid for the dinner, he announces the opposite of the actual result.
4. A cryptographer paid for the dinner iff the number of 'different' announcements is odd.

For example, a possible way in which the protocol may proceed is reflected in Figure 8.1. Then:

- A announces that he sees two different coins.

[1] This involves agents computing a function over inputs known only to them, without revealing any more information about the inputs than would be revealed from the output. For example, suppose Alice and Bob have one bit each and want to compute boolean AND. If they both have 1s, the output will be 1 and thus each will know the other's bit – this is considered OK. However, if Alice has a 0 and they find that the output is 0, Alice should not be able to tell if Bob has a 0 or a 1. (Again, if Bob has a 1, then he knows Alice has a 0 – this is also considered OK.)

- B announces that he sees two same coins.
- C sees two different coins, but being the payer announces that he sees two same coins.

From this, the the number of 'different' announcements is 1, which is odd, so A and B can conclude that one of the cryptographers paid for the dinner (of course, C already knows this). (We assume that all parties are cooperative and follow the protocol.) However, consider that it is not possible for either A to determine whether B or C paid:

- B flipped a fair coin, which landed either H or T.
- If it landed H, as in Figure 8.1, then B did not invert the difference, so C is the payer.
- The coin might have landed T, however, in which case given B's and C's announcements B inverted the difference and thus would have been the payer.
- For A to make any further conclusions he would need more information about the result of the coin flip, but the protocol does not allow for this.

It is also not possible for B to determine whether A or C paid:

- C flipped a fair coin, which landed either H or T.
- If it landed T, as in Figure 8.1, then A did not invert the difference, so C is the payer.
- The coin might have landed H, however, in which case A inverted the difference and C did not, so A is the payer.
- For B to make any further conclusions he would need more information about the result of the coin flip, but again the protocol does not allow for this.

However, they can deduce *someone* paid:

- Suppose no one paid, then everyone must have told the truth.
- Since B sees two same coins, A's coin flip must equal B's own coin flip.
- Since C sees two same coins, B's coin flip must equal C's own coin flip.
- But then A's coin flip must equal C's coin flip (by transitivity of equals), yet A said different – a contradiction.

As demonstrated above, there are intuitive logical arguments for the correctness of this protocol, as in [27]. Nonetheless, the problem also naturally lends itself to formulation as an interpreted system; we can express assertions that we wish to make about the protocol using temporal-epistemic logic. We encode this protocol into an interpreted system, as follows.

- Each of the three cryptographers is encoded as an agent, with local state corresponding to whether he is the payer and the two coins he sees. The agents' actions involve making announcements as to whether they see same or different coin flips.
- The agents' protocols require them to follow the overall protocol as described above.
- We restrict the initial states to those where either zero or one cryptographer(s) paid for the dinner.

- We define as atomic propositions $c_i paid$ for each cryptographer i, holding iff cryptographer i paid the bill, and *odd* and *even* corresponding to whether the total number of 'different' announcements was odd or even respectively.

We then express the interpreted system in ISPL (see Section 2.8.3), in particular using ISPL's `Obsvars`[2] construct for the coins, which allows for easier enforcement of the coins being consistent across agents. This allows us to express several properties of interest, which we used for our tests. It is worth noting that these specifications can be expressed in CTLK, LTLK and LDLK. Of course, they can also be expressed in CTL*K and CDL*K, though our algorithms for these logics will simply invoke the model checking algorithms for the linear time versions of the relevant logics; we would expect the results for these logics to be very similar to that for LTLK and LDLK respectively.

1. **If the number of 'different' announcements is odd, a non-paying cryptographer knows someone else paid, but does not know who.** This may be expressed in CTLK as

$$\bigwedge_{i=1}^{N} \left(AG \left((odd \wedge \neg c_i paid) \rightarrow \left(K_{Crypt_i} \left(\bigvee_{j \neq i} c_j paid \right) \wedge \bigwedge_{j \neq i} \neg K_{Crypt_i} c_j paid \right) \right) \right)$$

and equivalently in LTLK as

$$\bigwedge_{i=1}^{N} \left(G \left((odd \wedge \neg c_i paid) \rightarrow \left(K_{Crypt_i} \left(\bigvee_{j \neq i} c_j paid \right) \wedge \bigwedge_{j \neq i} \neg K_{Crypt_i} c_j paid \right) \right) \right)$$

and, finally, in LDLK as

$$\bigwedge_{i=1}^{N} \left([\top^*] \left((odd \wedge \neg c_i paid) \rightarrow \left(K_{Crypt_i} \left(\bigvee_{j \neq i} c_j paid \right) \wedge \bigwedge_{j \neq i} \neg K_{Crypt_i} c_j paid \right) \right) \right)$$

We use the symmetry of the problem and focus on cryptographer 1 rather than all cryptographers. The specification in CTLK then becomes

$$AG \left((odd \wedge \neg c_1 paid) \rightarrow \left(K_{Crypt_1} \left(\bigvee_{j \neq 1} c_j paid \right) \wedge \bigwedge_{j \neq 1} \neg K_{Crypt_1} c_j paid \right) \right)$$

and that in LTLK becomes

$$G \left((odd \wedge \neg c_1 paid) \rightarrow \left(K_{Crypt_1} \left(\bigvee_{j \neq 1} c_j paid \right) \wedge \bigwedge_{j \neq 1} \neg K_{Crypt_1} c_j paid \right) \right)$$

and finally in LDLK becomes

$$[\top^*] \left((odd \wedge \neg c_1 paid) \rightarrow \left(K_{Crypt_1} \left(\bigvee_{j \neq 1} c_j paid \right) \wedge \bigwedge_{j \neq 1} \neg K_{Crypt_1} c_j paid \right) \right)$$

[2] These are environment variables that are visible only to a subset of the agents.

2. **Under no circumstances can a cryptographer know that anyone other than themselves has paid.** This may be expressed in CTLK as

$$AG\left(\bigwedge_{i=1}^{N}\bigwedge_{j\neq i}(\neg K_{Crypt_i}c_j paid)\right)$$

and equivalently in LTLK as

$$G\left(\bigwedge_{i=1}^{N}\bigwedge_{j\neq i}(\neg K_{Crypt_i}c_j paid)\right)$$

and also in LDLK as

$$[\top^*]\left(\bigwedge_{i=1}^{N}\bigwedge_{j\neq i}(\neg K_{Crypt_i}c_j paid)\right)$$

3. **If the number of 'different' announcements is even, it is common knowledge that no one paid.** Where Γ refers to all of the cryptographers, this may be expressed in CTLK as

$$AG\left(even \rightarrow C_\Gamma\left(\bigwedge_{j=1}^{N}\neg c_j paid\right)\right)$$

and equivalently in LTLK as

$$G\left(even \rightarrow C_\Gamma\left(\bigwedge_{j=1}^{N}\neg c_j paid\right)\right)$$

and also in LDLK as

$$[\top^*]\left(even \rightarrow C_\Gamma\left(\bigwedge_{j=1}^{N}\neg c_j paid\right)\right)$$

These may be suitably expressed in the `Formulae` section of the relevant ISPL file. For example, for LTLK and 5 cryptographers, we can express the properties as follows.

```
Formulae
  -- property (1)
  LTL G((odd and !c1paid) ->
    (K(DinCrypt1,(c2paid or c3paid or c4paid or c5paid))
     and !K(DinCrypt1,c2paid) and !K(DinCrypt1,c3paid)
     and !K(DinCrypt1,c4paid) and !K(DinCrypt1,c5paid)));

  -- property (2)
  LTL G(!K(DinCrypt1, c2paid)
        and !K(DinCrypt1, c3paid)
        and !K(DinCrypt1, c4paid)
        (...)
        and !K(DinCrypt5, c4paid));

  -- property (3)
  LTL G(even ->
        GCK(cryptos, !c1paid and !c2paid and !c3paid and !c4paid and !c5paid));
end Formulae
```

(Naturally, these were programmatically generated; this would be essential for larger values of N.)

As far as acceptance testing is concerned, all of these properties should hold for any value of $N \geq 3$. That said, the focus of this section is more on performance testing, examining how well the LTLK and LDLK algorithms scale against the existing CTLK algorithm. Our experimental results for the aforementioned specifications are presented in Table 8.1. Note that the "Non-MC" time (the second column in each table) refers to the mean time taken by MCMAS to parse the input ISPL file, encode the model with BDD variables and compute the set of reachable states. Also, we have presented separately the results for a naïve implementation of the algorithm discussed in Section 5.1 (LDLK), and for said algorithm after applying the performance optimisations discussed in Section 5.4 (LDLK(OP)). We can make several observations about the data:

- Expectedly, in general it appears exponentially more time is needed to verify specifications for larger values of N (regardless of specification language), though there is a notable exception with $N = 10$ taking much longer than $N = 11$ and, to a lesser extent, $N = 11$ taking longer than $N = 12$.

  Note that the size of the model grows exponentially as N increases (consider that each additional cryptographer introduced adds one more coin), which makes even the CTLK algorithm require exponentially more time.

- Model checking LTLK specifications does indeed take longer than CTLK specifications; this makes sense in terms of the computational complexity of the relevant algorithms (linear in the size of the specification for CTLK but exponential for LTLK) as well as because there is a clear overhead in constructing the tableau and consistency rules. Running this through a profiler (we used `callgrind`) seemed to reveal that using the tableau method seems to require significantly many more iterations and hence more existential preimages than checking the equivalent CTLK specification.

- Depending on the specification, the *optimised* LDLK algorithm may perform notably better (specification 1), or similarly (for specifications 2 and 3) to the LTLK algorithm. We attribute this to our performance optimisations discussed in Section 5.4; observe that the unoptimised LDLK algorithm quickly runs out of memory and/or time, as the Büchi automata generated can be very large.

  The asymptotic complexity of our LDLK algorithm is actually marginally better than that of our LTLK algorithm ($O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$ for LDLK, and $O(2^{|\phi| \log |\phi|}|\mathcal{I}|)$ for LTLK). However, we suspect that the LTLK algorithm has potential to be optimised further (for instance, by using alternative variable encoding techniques as suggested in [80]). We did not focus on optimising the LTLK algorithm at all, apart from verifying propositional formulae over states as opposed to over paths.

- Considering the algorithms' performance as N increases, it seems that the LTLK algorithm is slowest on specification 1; the CTLK and optimised LDLK algorithms appear slowest on specification 2. The result for LTLK is interesting; we would expect performance to degrade most quickly on specification 2 owing to its quadratic growth in size as N increases (as compared to linear). Using `callgrind`, we traced this down to CUDD taking longer to compute *each* existential preimage when model checking specification 1. Owing to limited

Specification 1

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	LTLK	LDLK	LDLK(OP)	CTLK	LTLK	LDLK	LDLK(OP)
3	0.001	0.001	0.003	TIMEOUT	0.002	9368000	9748160	TIMEOUT	9649952
4	0.026	0.001	0.003	TIMEOUT	0.002	9360800	9802336	TIMEOUT	9667296
5	0.066	0.002	0.010	–	0.005	9517280	10906176	–	10241184
6	0.107	0.002	0.010	–	0.005	9608288	11029920	–	10291264
7	0.293	0.006	0.031	–	0.015	10581152	14161472	–	12330208
8	0.564	0.005	0.034	–	0.021	11183680	15233520	–	13610016
9	0.594	0.005	0.052	–	0.020	11943776	18780576	–	14235040
10	6.211	0.349	1.051	–	0.582	47320192	54309296	–	57092256
11	2.366	0.041	0.241	–	0.127	20551808	35235776	–	24236576
12	3.383	0.019	0.100	–	0.061	16028736	25104304	–	20008032
15	26.153	0.370	1.157	–	0.709	50538048	47728160	–	48033536
18	67.049	0.573	1.808	–	1.010	49998752	50665824	–	59538048
20	546.151	20.860	72.342	–	43.533	206514656	509806192	–	355785216
22	1,615.210	25.655	99.862	–	69.491	321949600	786076336	–	569407152
25	TIMEOUT	–	–	–	–	–	–	–	–

Specification 2

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	LTLK	LDLK	LDLK(OP)	CTLK	LTLK	LDLK	LDLK(OP)
3	0.005	0.001	0.002	TIMEOUT	0.002	9368000	9543552	TIMEOUT	9649952
4	0.027	0.001	0.003	OOM	0.003	9328064	9594704	OOM	9667296
5	0.067	0.003	0.006	–	0.006	9550016	10053424	–	10175712
6	0.109	0.002	0.008	–	0.006	9641024	10319984	–	10324000
7	0.296	0.008	0.023	–	0.020	10617984	12201104	–	12362944
8	0.561	0.009	0.035	–	0.024	11257344	13409200	–	13610016
9	0.600	0.011	0.047	–	0.029	12044032	14707856	–	14627940
10	6.260	0.473	0.694	–	0.689	45920576	52829760	–	56116384
11	2.359	0.066	0.168	–	0.133	20738016	28235104	–	24242720
12	3.399	0.043	0.125	–	0.075	15721824	20865984	–	20269952
15	30.217	0.542	0.893	–	0.699	50053152	39635552	–	39994688
18	77.260	0.572	1.347	–	1.152	49998752	60217184	–	59513472
20	585.183	24.741	46.994	–	48.275	206514656	300752192	–	355785216
22	1,808.148	28.713	67.472	–	74.346	321953696	416069552	–	569407152
25	TIMEOUT	–	–	–	–	–	–	–	–

Specification 3

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	LTLK	LDLK	LDLK(OP)	CTLK	LTLK	LDLK	LDLK(OP)
3	0.005	0.001	0.001	0.033	0.002	9368000	9620480	12374112	9649952
4	0.026	0.001	0.002	0.966	0.002	9360800	9643968	35088384	9667296
5	0.065	0.002	0.007	TIMEOUT	0.006	9582752	10250592	TIMEOUT	10273920
6	0.109	0.002	0.007	–	0.005	9641024	10366144	–	10324000
7	0.293	0.006	0.020	–	0.016	10650720	12720192	–	12395680
8	0.558	0.005	0.025	–	0.023	11290080	13764672	–	13421760
9	0.600	0.006	0.029	–	0.023	12078816	15140672	–	14668832
10	6.198	0.312	0.743	–	0.614	39422656	60352896	–	57135264
11	2.358	0.045	0.137	–	0.113	20807584	30065216	–	28404608
12	3.353	0.032	0.085	–	0.068	15881440	20504512	–	20546240
15	27.535	0.413	0.793	–	0.699	49889472	49820768	–	39978304
18	69.606	0.589	1.071	–	1.066	50064288	50204432	–	59579008
20	576.043	21.169	50.431	–	45.810	206514656	331828416	–	355785216
22	1,776.829	26.475	67.405	–	73.167	321953696	489645040	–	569407152
25	TIMEOUT	–	–	–	–	–	–	–	–

Table 8.1: Experimental results for the Dining Cryptographers scenario.

time as well as a lack of understanding of the internals of CUDD, we decided not to investigate this point further.

- In terms of memory, we observe that verifying CTLK properties was generally cheapest (with an exception when $N = 15$, which may have arisen when computing intermediate sets of states). Again, depending on specification as well as N, either LTLK or LDLK properties were the most expensive. CTLK using the least memory is intuitive, owing to the complexity of the additional symbolic structures we construct when verifying LTLK or LDLK properties. For example, consider Specification 1.

  - For LTLK, the tableau construction requires an additional $2(N + 2)$ BDD variables; two for each of the N epistemic modalities, as well as two for each of the atomic propositions *odd* and c_1*paid*.
  - For LDLK (optimised), the automaton we generate always uses 18 additional BDD variables. This is because we are constructing automata for formulae of the form $\langle\top^*\rangle\neg\psi$ for some subformula ψ which is only dependent on the current state. We thus apply propositional shortcircuiting and avoid constructing the automaton for ψ. The resulting alternating automaton has 5 states (2 from the regex automaton, after minimising the ϵ-NFA, and 3 from the automaton for a propositional formula). Only one of these states (the state after the true transition from the propositional formula) is accepting, and we thus have $4 \times 4 + 2 = 18$ variables. Note that the amount of additional memory used is more than constant, because the transition relation of the alternating automaton also uses the variables in the original model.

- Interestingly, for CTLK it seems that peak BDD memory usage is equal across two or all three specifications in several cases (such as specifications 1 and 3 for $N = 4$; all three for $N = 3$ and $N = 20$). We suspect that the CTLK algorithm might be even more efficient for those cases; this observation suggests that in those cases, peak memory usage was involved at the time of building the model and/or computing the set of reachable states. This is, unsurprisingly, not the case for LTLK, since we need to create the relevant tableau for the formula and compose it with the model, and these tableaus have different sizes even for the same N.

  We also observe equal peak BDD memory usage across the specifications for the optimised LDLK algorithm (for $N = 3$, $N = 20$ and $N = 22$) – we suspect the meaning here is different, however, since building the model or computing reachable states is independent of our formulae and these values are greater than the corresponding values for CTLK. We have established above that the number of additional BDD variables used in the last step is constant. Furthermore, recall that our algorithm for checking $[\top^*]\psi$ requires us to construct the alternating automaton for $\langle\top^*\rangle\neg\psi$. However, because our specifications are true, $\neg\psi$ is actually unreachable in the model, and the final alternating automata we build for all of the specifications are isomorphic to one another. Thus, for these values of N we suspect that peak BDD memory usage occurred during the symbolic breakpoint construction – this is a deterministic procedure and the resulting Büchi automata would thus also be isomorphic. (Note that runtimes still differ, perhaps owing to different times taken to show that $\neg\psi$ indeed is unreachable in our model.)

- It appears that although LTLK and (optimised) LDLK model checking is slower than CTLK model checking, the overall execution time is still dominated by other parts of MCMAS, in particular building the model and computing the set of reachable states as mentioned

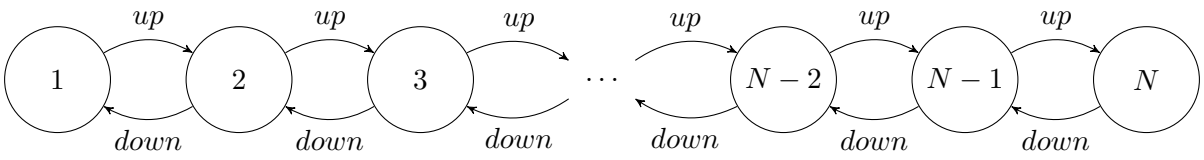


Figure 8.2: State space of a counter for arbitrary N.

in Section 2.8.2. With that in mind, this slowdown seems acceptable, and in most cases relatively insignificant.

8.5 Counter

Consider an integer counter which ranges from 1 to some *even* positive integer N. The counter changes its value by 1 on each time step (it can non-deterministically increment or decrement its value, as long as it stays within the allowable range). This can be encoded into a simple interpreted system as follows:

- We have a single agent, which has a state corresponding to the value of the counter. The agent has two actions *up* and *down*.
- The agent's protocol is used to ensure that the value of the counter stays in range. (Thus, we permit *up* only when the counter's value is not N, and permit *down* only when the counter's value is not 1.)
- The transition relation simply updates the counter's local state appropriately (increasing the value by 1 if the action is *up*, and decreasing it by 1 if the action is *down*).
- We restrict the initial states to those in which the value of the counter is even.
- We define several atomic propositions of interest:
  - *even*, which holds in all even states;
  - *max*, which holds when the counter's value is at its maximum (i.e. N), and
  - *nearMax*, which holds when the counter's value is either 1 below its maximum, or at its maximum (i.e. $N-1$ or N).

This is more intuitively represented in Figure 8.2. We can then express several properties about the counter's behaviour:

1. **After an even number of steps, the value of the counter is even.** Intuitively, this is true owing to the parity of the counter after an even number of operations. This can be expressed as the LDLK formula
$$[(\top;\top)^*]\mathit{even}$$
2. **After an odd number of steps, the value of the counter is odd.** Again, this is true owing to the parity of the counter after an odd number of operations. This can be expressed as the LDLK formula
$$[\top;(\top;\top)^*]\neg\mathit{even}$$

3. **It is always the case that if the value of the counter is at its maximum, then after every two steps the counter will still be at its maximum**. This is false, except for $N = 2$, since one can perform two *down* actions. This can be expressed as the LDLK formula
$$[\top^*](max \to [(\top;\top)^*]max)$$

4. **It is always the case that if we are in an even state, then it is possible to reach the maximum state following an *alternating sequence* of even and odd states.** Intuitively, this holds; the sequence must alternate between even and odd, as a result of parity. This can be expressed as the CDL*K formula
$$A[\top^*](even \to E(\langle(even;\neg even)^*\rangle max))$$

5. **If the value of the counter starts near maximum, then there exists a path which is infinitely often *not* near maximum, yet is near maximum after every block of three steps.** Again, this is intuitively true as long as $N > 2$. Observe that if the counter is at $N-1$ we can perform *down*, *up*, *up* which leaves the counter at N; if the counter is at N we can perform *down*, *down*, *up* which leaves the counter at $N-1$. This path also visits $N-2$ infinitely often. This can be expressed as the CDL*K formula
$$nearMax \to E([\top^*]\langle\top^*\rangle(\neg nearMax) \wedge [(\top;\top;\top)^*]nearMax)$$

We present our experimental results for verifying the five aforementioned specifications over counters of varying size in Table 8.2, using the optimised LDLK and CDLK algorithms.

	Non-MC	Model Checking Time (s)					BDD Memory (bytes)				
N	Time (s)	Spec 1	Spec 2	Spec 3	Spec 4	Spec 5	Spec 1	Spec 2	Spec 3	Spec 4	Spec 5
2	0.002	0.001	0.002	0.030	0.003	0.008	9037520	9157008	11165072	9330512	9884176
4	0.002	0.001	0.002	0.055	0.002	0.046	9074352	9161104	12129616	9355888	12227824
16	0.002	0.001	0.001	0.049	0.004	0.064	9103984	9202192	11759440	9429712	12461264
64	0.002	0.001	0.002	0.070	0.007	0.054	9177968	9276176	12068720	9606000	12535248
256	0.009	0.001	0.002	0.148	0.027	0.072	9567056	9655264	12639920	10281552	13601616
1024	0.136	0.004	0.005	0.509	0.155	0.074	9921328	9921328	15087440	11059408	13771760
4096	1.534	0.324	0.342	5.930	2.904	0.033	11130992	11229200	26486928	14110576	13268752
8192	7.909	1.462	1.484	15.327	15.008	0.065	15835280	15933488	26613936	23692944	17975152
16384	22.720	5.007	5.742	14.391	8.080	0.040	13734000	13734000	28142608	20214640	15906608
32768	108.104	19.276	20.134	32.389	33.516	0.037	20480272	20480272	35689073	24614096	20635472
65536	412.164	76.708	82.539	294.255	533.740	0.144	34120944	34120944	40447312	46335376	33882928
131072	2,171.876	859.138	918.807	110.722	936.337	0.072	49532208	49532208	50107248	49874576	49685360
262144	12,068.591	4,632.815	4,748.332	275.927	5,079.043	0.108	62289040	62289040	62569168	62402032	62442192
524288	49,737.791	19,926.556	20,812.064	17,651.938	47,158.764	1.747	95112592	95112592	94755792	95454960	94677968
1048576	TIMEOUT	–	–	–	–	–	–	–	–	–	–

Table 8.2: Experimental results for the Counter scenario.

From these results, we draw several conclusions:

- In general, for larger values of N, it takes longer to verify the various specifications. We observe that there are a few exceptions to this (specification 3, $N = 131072$ and 262144; specification 4, $N = 16384$; many of the results for specification 5).
- For each value of N, specification 2 generally takes slightly longer to verify than specification 1. We expect this owing to the (slightly) larger alternating automaton for specification 2; we add one more state to the ϵ-NFA for the regular expression, which is non-accepting since we need to construct the automaton for $\langle\top;(\top;\top)^*\rangle even$. This results in MCMAS using 4 more BDD variables; the increase is marginal (certainly not by a factor of $2^4 = 16$), presumably owing to the empirical efficiency of BDDs.
- For smaller values of N, as expected, the memory used for BDDs is slightly larger for specification 2 than for specification 1. However, for $N \geq 16384$ we observe that they are equal, and unlike in the Dining Cryptographers scenario (discussed in Section 8.4) the alternating automata are certainly *not* isomorphic to one another. Nonetheless, the peak memory usage is the same; we suspect this may be a result of the formula being unsatisfiable in both cases (since both specifications are true).
- Specification 3 initially took the longest to verify, though it seemed to scale relatively well. While the specification does seem larger than specifications 1 or 2, profiling reveals that the construction of the alternating automaton is nonetheless much faster for specification 3, perhaps because the predicate max applies to just 2 specific states. We observed that for the same value of N, it seems more iterations are needed before the algorithm for checking $EG\top$ converges; this initially dominates runtime, though for larger values of N the cost of automata construction for specifications 1 and 2 overtakes this.
- Specification 4 consistently took the longest time to verify; this is unsurprising, seeing as verifying it first requires constructing an automaton for $\langle(even;\neg even)^*\rangle max$ and checking it with the model. This automaton is similar in structure to the automaton we build for specification 1. We then need to construct another automaton for $\langle\top^*\rangle(even \wedge \neg\psi)$ where ψ holds in the states that $E\langle(even;\neg even)^*\rangle max$ holds, and check it with the model.
- On the other hand, Specification 5 was, for larger values of N, by far the quickest to verify. There is only one path modality, though model checking $E([\top^*]\langle\top^*\rangle(\neg nearMax) \wedge [(\top;\top;\top)^*]nearMax)$ was still surprisingly quick. Firstly, we observed that constructing the automaton was fast, probably because $nearMax$ applies to two specific states. Then, checking whether a given path satisfies $[\top^*]\langle\top^*\rangle(\neg nearMax) \wedge [(\top;\top;\top)^*]nearMax$ is also fast:
  - If a path starts in a state *not* near the maximum, it will immediately be rejected since it will not satisfy $[(\top;\top;\top)^*]nearMax$.
  - If a state is near the maximum, there is a path with a period of just length 6 that will satisfy the formula.

  In other words, for every path we can determine whether it satisfies $([\top^*]\langle\top^*\rangle(\neg nearMax) \wedge [(\top;\top;\top)^*]nearMax)$ relatively quickly – intuitively, only a constant number of iterations is required before our fix-point algorithm for checking $EG\top$ converges. We profiled several runs using `callgrind` and observed that, regardless of N, precisely 28 calls to `check_EX`, the method for finding the successor states of the system, were made. Hence, the model checking step is fast, though generally still increasing with time as the time taken to compute each preimage may have increased with increasing N.

8.6 Bit Transmission

Recall the Bit Transmission Protocol, presented earlier in Section 2.2.2, where the Sender aims to communicate the value of a bit to the Receiver over a channel that may drop messages. We can readily encode this system in ISPL (see Appendix A for a possible implementation).

Observe that the state space of this interpreted system is small:

- The Sender has 4 possible states; it has a bit, which is either 0 or 1, and a boolean indicating whether it has received an ack or not.
- The Receiver has 3 possible states; it either has not received the bit yet, has received a 0, or has received a 1.
- The Environment has 4 possible states; it can send or drop messages in each direction (sender to receiver, and vice versa).

This suggests 48 possible states. However, not all of these states are reachable; the actual number of reachable states is 22.[3] The purpose of this scenario is, then, to consider how our algorithms scale with the size of the formulae being verified. In terms of computational complexity, this is a theoretical weakness of both our LTLK and LDLK algorithms – they require time exponential in the size of the formula.

We thus verify the following families of specifications over our model, for differing values of N and differing choices of algorithms (depending on the languages in which properties can be specified).

1. **It is possible for the Sender to receive the ack within N steps.** This is true for $N \geq 2$ (consider the situation where the environment always works). This can be expressed in CTL as
$$\underbrace{EX \dots EX}_{N \text{ times}} recack$$
It can be expressed in CTL* as
$$\underbrace{E(X(\dots E(X}_{N \text{ times}}(recack)\dots))$$
It can be expressed in CDL* as
$$\underbrace{E([\top](\dots E([\top]}_{N \text{ times}}(recack)\dots))$$
Of course, this can be equivalently expressed in CTL* and CDL* with only one path quantifier – i.e. $E(X \dots X recack)$ or $E[(\top; \dots; \top)]recack$, but for this specific formula, there is no need to reason over paths.
2. **On every path, if the environment is working for the first N steps, then the ack is definitely received after said N steps.** This is true for $N \geq 2$, but not for

[3] Considering the joint states of Sender and Receiver, there are actually only 6 possibilities: $\{(0,?),(1,?),(0,0),(1,1),(0_A,0),(1_A,1)\}$ where ? denotes "no information" and 0_A denotes "0 with ack received" (and 1_A is defined similarly). Also, the ISPL implementation remembers the previous environment state, so $(0,0)$ and $(1,1)$ after a delivery to the Sender but not the Receiver are also unreachable.

$N = 1$ (since only one message has been sent at this point). This cannot be expressed in CTL as it considers structural properties of specific paths. It can be expressed in LTL as

$$\underbrace{X(envworks \wedge X(envworks \wedge X(...)))}_{\text{depth of } N \text{ } X\text{s}} \rightarrow \left(\underbrace{X \ldots X}_{(N+1) \text{ times}} \; recack \right)$$

and in LDL as

$$[\top; \underbrace{envworks; \ldots; envworks}_{N \text{ times}}] recack$$

Note that this property can also be specified as the following LTL specification, though we tested the specification with nesting of Xs as the size of the above specification is linear as opposed to quadratic in N.

$$\left(\bigwedge_{i=1}^{N} \underbrace{X \ldots X}_{i \text{ times}} envworks \right) \rightarrow \left(\underbrace{X \ldots X}_{(N+1) \text{ times}} \; recack \right)$$

We tested the appropriate algorithms on the specifications above; in the LDL or CDL* cases, we tested both with and without optimisation. Furthermore, for Specification 2, we tested the effect of applying only the automata simplification optimisations (i.e. using the naïve method of computing the symbolic breakpoint construction) – this is shown in the table as CDL*K(ASO) meaning *automata simplification only*, while CDL*K(OP) refers to the CDL*K algorithm with all optimisations included.

Our experimental results are presented in Table 8.3.

Specification 1

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	CTL*K	CDL*K	CDL*K(OP)	CTLK	CTL*K	CDL*K	CDL*K(OP)
5	0.002	0.000	0.001	0.002	0.002	9018048	9158816	9398208	9398208
10	0.002	0.000	0.002	0.004	0.004	9018048	9299584	9819296	9819296
20	0.002	0.000	0.003	0.009	0.009	9018048	9591360	10686368	10686368
30	0.002	0.001	0.005	0.013	0.013	9018048	9887232	11508176	11508176
40	0.002	0.000	0.007	0.018	0.018	9018048	10187200	12391952	12391952
50	0.003	0.000	0.009	0.023	0.023	9018048	10476928	13258288	13258288
100	0.002	0.001	0.019	0.058	0.058	9018048	11989856	25937664	25937664
250	0.002	0.002	0.068	0.154	0.156	9018048	24832288	38454768	38454768
500	0.003	0.004	0.156	0.351	0.362	9018048	32484256	60201136	60201136
750	0.003	STACK OVERFLOW							

Specification 2

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTL*K	CDL*K	CDL*K(ASO)	CDL*K(OP)	CTL*K	CDL*K	CDL*K(ASO)	CDL*K(OP)
2	0.003	0.001	0.004	0.003	0.002	9191552	9600352	9311040	9376512
3	0.002	0.002	0.019	0.006	0.005	9313168	10820672	9542240	9687520
4	0.002	0.005	2.099	0.020	0.014	9763552	17364224	10150752	10543616
5	0.003	0.023	804.672	0.138	0.040	12219184	54799968	12023008	13152480
6	0.003	0.140	55,017.928	4.999	0.150	26329632	268374624	17525856	20859136
7	0.003	1.965	TIMEOUT	85.651	8.169	93538864	TIMEOUT	34936832	44230496
8	0.003	15.642	–	1,031.127	152.041	468454240	–	94779200	99197152
9	0.003	215.265	–	11,632.178	4,653.998	2717969456	–	260908768	273472480
10	0.006	681.687	–	TIMEOUT	70,085.323	15737761440	–	TIMEOUT	794670688
11	0.005	OOM	–	–	TIMEOUT	OOM	–	–	TIMEOUT

Table 8.3: Experimental results for the Bit Transmission scenario.

From these results, we observe the following:

- In general, the time taken to verify specifications increases as N increases. This is in line with expectations, as all of the algorithms are at least linear (if not exponential) in the size of the formulae being verified.

- For specification 1, the CTLK algorithm is unsurprisingly the fastest; the CTL*K and CDL*K algorithms (in that order) require more time and memory, as they need to construct additional symbolic structures which are composed with the model. The peak BDD memory consumption of the CTLK algorithm is constant, suggesting that this was incurred when building the model. Nonetheless, all of the algorithms are able to verify large specifications reasonably quickly.

- The optimisations we introduced for the CDL*K algorithm in Section 6.4.5 do not seem to be helpful for Specification 1; in fact, for larger N we observe a small but notable drop in performance for the optimised algorithm. This is in line with expectations, as Specification 1 does not admit any of the automata simplification optimisations we introduced, and there is a small overhead involved in checking *whether* an optimisation is applicable (such as attempting to prune unreachable states from ϵ-NFAs). The automata for Specification 1 are also very small, limiting the extent to which efficient conjunct computation could help boost performance.

- We encountered stack overflows when attempting to verify Specification 1 with large values of N (such as $N = 750$). This is a result of MCMAS's recursive implementation of how modal formulae are checked (`check_formula()`), which assumes that formulae can be dealt with recursively. Fixing this would require a significant amount of effort to refactor the existing code to use an iterative approach (such as simulating the call stack on a suitable heap-allocated data structure). It would also have questionable practical use (formulae of depth 500 were being dealt with fine); we thus decided not to focus on it.

  - It is worth noting that although the CTL*K and CDL*K formulae have a nesting depth of $2N$, our algorithms recurse only on the inner path quantifier (i.e. each step actually travels 2 "levels" in). Thus, even though these formulae have double the nesting depth as the CTLK formula for the same N, we do not see the stack overflow as early as $\frac{750}{2} = 375$ (we were able to check $N = 500$ for these specification languages as well).

- For Specification 2, the CTL*K algorithm scales better than the (fully) optimised CDL*K algorithm; although the CDL*K algorithm might actually seem to have better asymptotic complexity, we expect that the constant factors involved are considerably larger, and furthermore we are uncertain as to whether the relatively subtle difference between the complexities would be highlighted for small values of N.

  - As far as model checking time is concerned, both algorithms appear to scale exponentially in N for this specification; we plot this in Figure 8.3. For our (experimentally determined) model checking times T, we performed a linear regression of $\log T$ against $N \log N$, to investigate how well T follows our theoretical complexity result $2^{N \log N}$. It appears we have a good fit, with $R^2 = 0.9897$ for CTL*K and $R^2 = 0.9703$ for CDL*K.
  - We observed that our optimisations for CDL*K were also useful for this specification. There were two optimisations that were particularly relevant:

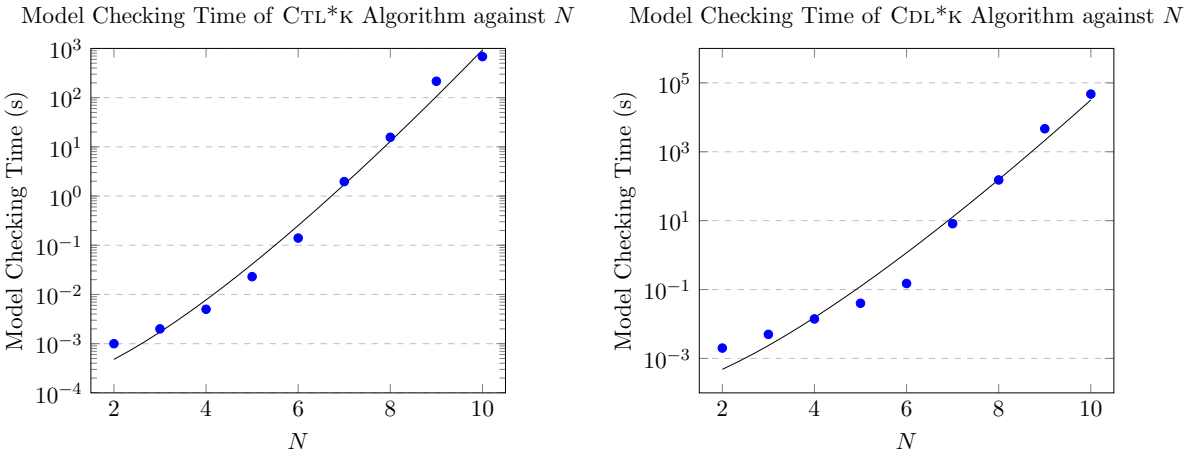


Figure 8.3: Plot of the model checking time required for Specification 2 for the Bit Transmission protocol, where the property is specified in CTL*K and CDL*K. Note the use of a logarithmic scale on the vertical axes.

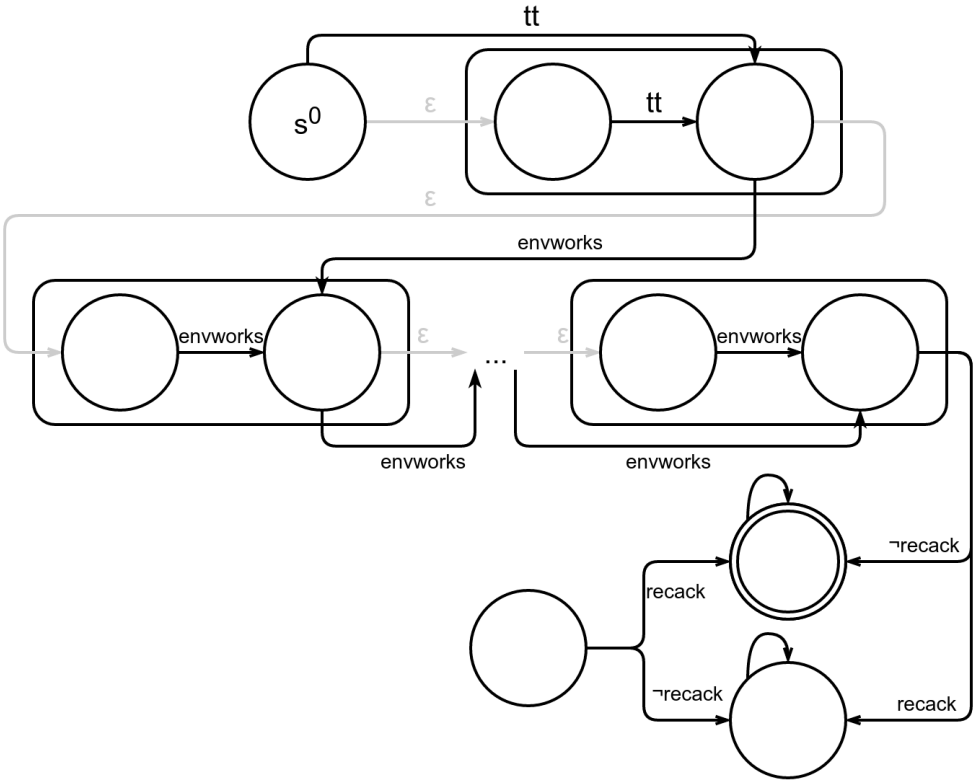
- Firstly, the pruning of states with no inbound non-ϵ transitions from the ϵ-NFAs was effective. Consider Figure 8.4, which shows why this optimisation is very useful in this case; we use $4N + 4$ fewer BDD variables (which leads to an exponential reduction in the size of the state space that we subsequently need to consider).
- Secondly, the efficient conjunct computation (Section 5.4.4) optimisation is also very relevant, because the automata for these formulae are relatively complex. We observe a significant speed-up by adding this optimisation, and also become able to verify the specification for $N = 10$ for which automata simplification alone appeared insufficient. Interestingly, we actually use (slightly) more memory when this optimisation is active; since this is a measure of *peak* BDD memory usage, it is possible that our accumulator reaches an intermediate state which uses more memory than the final result of $t_I \wedge t_{LC} \wedge t_R$.

– However, the CTL*K algorithm appears substantially more memory-hungry than the optimised CDL*K algorithm. We suspect that this might be owing to the way we build consistency rules for the tableau construction (discussed in Section 3.2.3); our implementation currently separately constructs these consistency rules for each occurrence of *envworks*. This could perhaps be addressed by "canonicalising" atomic propositions – that is, constructing a *single* tableau variable for each unique atomic proposition.

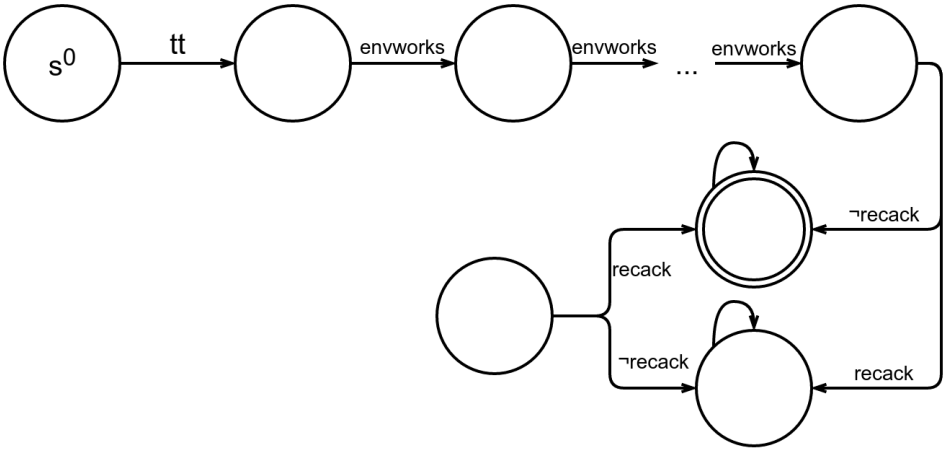
8.7 Prisoners

The *100 prisoners and a lightbulb* puzzle may be stated as follows, as in [92]:

> *A group of 100 prisoners, all together in the prison dining area, are told that they will be all put in isolation cells and then will be interrogated one by one in a room containing a light with an on/off switch. The prisoners may communicate with one another by toggling the light switch (and that is the only way in which they can communicate). The light is initially switched off. There is no fixed order of interrogation,*



(1) Unoptimised alternating automaton; this has $3 + (2 \times N) + 2 = 2N + 5$ rejecting states, and 1 accepting state. (We leave the ϵ-transitions that were generated by Thompson's construction and subsequently removed by Equation 5.2 in grey, to show why so many states were generated.) The number of BDD variables this contributes to the symbolic Büchi automaton is $4(2N + 5) + 2(1) = 8N + 22$.



(2) Alternating automaton, after pruning states which have only inbound ϵ-transitions in the ϵ-NFA (and rewiring states accordingly). This has $2 + N + 2 = N + 4$ rejecting states, and 1 accepting state; the number of BDD variables this contributes to the symbolic Büchi automaton is $4(N + 4) + 2(1) = 4N + 18$.

Figure 8.4: Representation of alternating automata for the LDL formula $\neg[\top; envworks; \ldots; envworks]recack$, where *envworks* occurs N times, before and after pruning states of the ϵ-NFA with only inbound ϵ-transitions.

> *or fixed interval between interrogations, and the same prisoner may be interrogated again at any stage. When interrogated, a prisoner can either do nothing, or toggle the light-switch, or announce that all prisoners have been interrogated.*
>
> *If that announcement is true, the prisoners will (all) be set free, but if it is false, they will all be executed. While still in the dining room, and before the prisoners go to their isolation cells, can the prisoners agree on a protocol that will set them free (assuming that at any stage every prisoner will be interrogated again sometime)?*

Clearly, the problem is generalisable to positive integer N prisoners. One possible solution to this problem involves the prisoners setting up a simple protocol where one of them serves as a *counter*, as discussed in [92] (the paper also discusses other more complex alternatives that may result in a faster correct announcement):

- Non-counting prisoners switch on the light the first time they enter the room with the light off; otherwise they do nothing.
- The counting prisoner keeps track of the number of times he has entered the room with the light on. For the first $N-2$ times this happens, he switches the light off; thereafter, he announces (correctly) that all prisoners have been interrogated at least once.

Intuitively, the protocol is correct as the counting prisoner must see the light switched on $N-1$ times, and each of these acts must have been done by a distinct non-counting prisoner. It also eventually reaches a conclusion since each prisoner must be interrogated infinitely often. We can encode the setup above into an interpreted system as follows:

- The *environment* keeps track of whether the bulb is on or off, as well as which prisoner is currently being interrogated (if any).
- We define one agent for the *prison*, which as an action chooses one of the prisoners to interrogate. The prison also keeps local state noting which prisoners have been interrogated, and whether a decision to *release* or *execute* the prisoners has been made.
- We define one agent for the *counter* (without loss of generality, we treat this as prisoner number 0). The counter keeps track of the number of times it has switched the light off; and, if chosen to be interrogated, either switches the light off and updates its count, does nothing (if the light is already off) or announces that all prisoners have been interrogated following the aforementioned protocol.
- The remaining prisoners are each represented as an agent following the protocol as discussed above. They need two local states, tracking whether they have been interrogated with the light off for the first time or not.
- We restrict the initial states to those where the bulb is off, the counter's internal count is 0, and the prison's internal state is that no one has been interrogated yet, and neither a release nor an execute action has taken place yet.
- We define as atomic propositions $interrogated_i$ for each prisoner i holding precisely in the states when prisoner i has been interrogated, *release* and *execute* holding in the states after an announcement has been made depending on whether or not all prisoners have actually been interrogated, *announce* holding after the counter has made an announcement, and *on* holding in the states where the lightbulb is on.

We can express the above system in ISPL, taking note of the following caveats:

- We define in the `Evaluation` section several relevant propositions corresponding to the aforementioned atomic propositions.
- We add *fairness constraints* (`Fairness`) that each prisoner is interrogated infinitely often, to encode the assumption that "at any stage every prisoner will be interrogated again sometime". However, since the prison stops interrogating prisoners after a decision to release or execute has been made, the actual fairness constraint is implemented as a disjunction of these (that is, `InterrogateI or Release or Execute` for each prisoner $0 \leq I < N$).
- The system follows a two-step cycle:
  1. The prison agent selects a prisoner to be interrogated (this choice is its action for this step). None of the other agents perform significant actions (i.e. their actions do not change the state of the system).
  2. The prisoner that has been selected may interact with the light bulb according to his protocol. If this is the counter, he may, on this step, also announce that all prisoners have been interrogated. If no announcement was made, go back to step 1.

We seek to verify several properties about the protocol, as follows:

1. **The protocol satisfies *liveness*; the prisoners will eventually be released.** This may be expressed in CTLK as $AF(release)$, in LTLK as $Frelease$, and in LDLK as $\langle \top^* \rangle release$. This formula is true for any value of N, because of the constraint that at any stage, every prisoner will be interrogated again sometime. (Note that without fairness constraints, it is possible for a prisoner to never be selected for interrogation and thus for the protocol to never terminate.)
2. **The counter knows that all prisoners have been interrogated when he announces this.** This property is true for any N, and may be expressed in CTLK as

$$AG\left(announce \rightarrow K_{Counter}\left(\bigwedge_{i=0}^{N-1} interrogated_i\right)\right)$$

   and equivalently in LTLK as

$$G\left(announce \rightarrow K_{Counter}\left(\bigwedge_{i=0}^{N-1} interrogated_i\right)\right)$$

   and also, equivalently in LDLK as

$$[\top^*]\left(announce \rightarrow K_{Counter}\left(\bigwedge_{i=0}^{N-1} interrogated_i\right)\right)$$

3. **The announcement is always first made on an even round.** This is true for any N, because of the way the system is implemented which follows a two-step cycle of choosing a prisoner to interrogate and allowing the interrogated prisoner to act. This may be expressed in LDLK as

$$\langle \top; (\top; \top)^* \rangle (\neg announce \wedge [\top] announce)$$

Notice that the atomic proposition *announce* becomes true forever once the announcement has been made, hence to verify that the announcement was *first* made on an even round, we need the formula to hold after matching the regular expression to be $\neg announce \wedge [\top]announce$. This property is not expressible in either CTLK or LTLK (or, for that matter CTL*K) as it discusses parity.

4. **It is possible for the bulb to be repeatedly toggled, until the prisoners are released.** This is a variant of the alternating sequences property (Specification 4 in Section 8.5), adjusted for the two-step cycle of our interpreted system. It is true for any N; the path where the prison alternates selecting an uninterrogated non-counting prisoner and the counter satisfies this property. This is expressible in CDL* as

$$E\langle(\neg on; \neg on; on; on)^*\rangle release$$

Like property 3, this property is not directly expressible in either CTLK or LTLK, because it is concerned with the structure of individual paths as well as with the existence of a suitable path. However, we can attempt to reason about it in CTL*K because we know that each non-counting prisoner can only switch the bulb on once. Hence, such a path must necessarily terminate in $4(N-1) = 4N-4$ steps (i.e. the path modality will match the $\neg on; \neg on; on; on$ pattern $N-1$ times). We can thus express the (English) property in CTL*K as follows (though, of course, this is not equivalent in the general case):

$$E\left(\left(\underbrace{\neg on \wedge X(\neg on \wedge X(on \wedge X(on \wedge X(\ldots))))}_{N-1 \text{ cycles; 1 cycle refers to a check over 4 steps}}\right) \wedge \left(\underbrace{X \ldots X}_{4N-4\ Xs} release\right)\right)$$

Notice that the size of this formula is linear in N, while the CDL*K formula is independent of N.

We present our experimental results in Tables 8.4 and 8.5. From the data, we draw the following conclusions:

- As before, the time and memory required to verify the various formulae increases as N increases. Note that the state space increases with $N \log N$ as N increases, since although we extend the Prison and Environment with a constant number of variables, as well as add a new non-counting prisoner agent, the counter's range must also increase. These increases in time and memory appear to be more than $N \log N$; given that this occurs not just for our algorithms but also for the base CTLK algorithm, we suspect this is a result of higher overheads involved in BDD manipulation. For the $N = 70$ case, it may be the case that CUDD's cache limit was reached as well.
  - Further to the above, it is in line with expectations that CTLK model checking is fastest; generally, this is followed by CDL*K and then CTL*K. In general, we attribute this to the performance optimisations introduced for CDL*K; we did not focus on optimising our CTL*K algorithm. However, this is not the case for specification 1, on which the CTL*K algorithm performs slower than even the *unoptimised* CDL*K algorithm. We profiled both the CTL*K and CDL*K algorithms on specification 1, and found that the CTL*K algorithm required between 1.6 to 1.8 times as many preimages when checking $EG\top$.
  - Generally, although the CTL*K and (optimised) CDL*K algorithms are slower than the CTLK algorithm, they are less than an order of magnitude slower. This supports

Specification 1

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	CTL*K	CDL*K	CDL*K(OP)	CTLK	CTL*K	CDL*K	CDL*K(OP)
3	0.003	0.003	0.011	0.010	0.009	9346720	9679008	9796448	9796448
4	0.014	0.006	0.025	0.017	0.017	9560160	10074560	9876896	9844160
5	0.034	0.022	0.094	0.054	0.052	10264752	11112688	11487952	11447024
6	0.072	0.025	0.121	0.064	0.061	10633632	12412544	11437344	11265472
7	0.154	0.053	0.232	0.118	0.113	11184624	12740528	12053808	11953552
8	0.284	0.096	0.413	0.196	0.192	12275888	14431312	13233040	12940464
9	0.425	0.177	0.757	0.333	0.330	13123744	15784576	15443680	15132672
10	0.799	0.284	1.211	0.523	0.507	14409296	16421520	16610544	16356816
12	1.714	0.580	2.601	1.007	0.969	18781040	28743984	29131472	20255888
15	6.896	1.641	8.054	2.705	2.758	39503904	48158048	45105664	45015648
20	37.718	5.473	30.722	9.339	8.769	58307168	58480000	58466464	58466464
25	97.090	14.526	92.619	26.049	25.596	58527088	58794128	58809264	58762160
30	283.561	46.038	333.814	102.538	93.654	59992608	60044688	60055888	60055888
35	720.700	94.541	561.454	174.097	166.856	95526112	95518960	95520080	95485264
40	1,855.243	193.303	1,303.565	392.683	377.383	167638608	167631616	167575472	167523072
45	4,094.211	613.567	4,090.987	1,366.970	1,319.548	239580192	239595808	239396464	239196448
50	4,107.177	767.768	4,143.739	1,139.734	1,127.406	271397200	271408880	271376112	271335152

Specification 2

	Non-MC	Model Checking Time (s)				BDD Memory (bytes)			
N	Time (s)	CTLK	CTL*K	CDL*K	CDL*K(OP)	CTLK	CTL*K	CDL*K	CDL*K(OP)
3	0.004	0.001	0.005	0.091	0.004	9346720	9531680	14448096	9520224
4	0.014	0.002	0.007	29.635	0.005	9560160	9581440	60731776	9602720
5	0.034	0.005	0.017	164,027.331	0.011	9959888	10410864	666474896	10352336
6	0.071	0.005	0.017	TIMEOUT	0.012	10637728	11074368	TIMEOUT	10993344
7	0.154	0.009	0.029	–	0.012	10973872	11502576	–	11392912
8	0.283	0.012	0.044	–	0.031	11928016	11955440	–	11974672
9	0.421	0.018	0.065	–	0.046	12655136	12690752	–	12703840
10	0.802	0.026	0.098	–	0.066	12991280	13761456	–	13416496
12	1.719	0.043	0.151	–	0.097	15580880	16702928	–	17076144
15	6.951	0.084	0.324	–	0.210	24397728	34904832	–	25430592
20	37.300	0.262	0.850	–	0.615	58008160	58054016	–	58093728
25	106.786	0.456	1.553	–	1.117	49694064	49782928	–	49918896
30	286.829	0.980	3.304	–	2.472	59984416	60073370	–	60094800
35	719.531	2.073	6.286	–	4.871	95444192	95506672	–	95546704
40	1,854.067	3.216	9.815	–	7.865	167501392	167635712	–	167600048
45	4,058.024	7.170	31.083	–	18.830	239281184	239464736	–	239482480
50	4,081.914	12.667	37.794	–	27.984	270987600	271034096	–	271036144
70	83,361.493	171.577	482.374	–	372.987	1056725152	1057175664	–	1057050736

Table 8.4: Experimental results for Specifications 1 and 2 for the Prisoners scenario. For Specification 2, $N = 5$, we report a result for unoptimised CDL*K that is longer than our 100,000 second timeout, as MCMAS-Dynamic completed model checking before we manually checked on the process.

Specification 3

N	Non-MC Time (s)	Model Checking Time (s) CDL*K	CDL*K(OP)	BDD Memory (bytes) CDL*K	CDL*K(OP)
3	0.004	TIMEOUT	0.064	TIMEOUT	12513280
4	0.015	–	0.082	–	13430624
5	0.035	–	0.119	–	14908688
6	0.072	–	0.161	–	16186016
7	0.157	–	0.261	–	19343824
8	0.289	–	0.304	–	16576080
9	0.430	–	0.517	–	19570336
10	0.813	–	0.767	–	20381168
12	1.771	–	1.454	–	36720336
15	6.551	–	3.339	–	47253760
20	39.149	–	10.460	–	58631904
25	97.479	–	31.883	–	62000736
30	281.561	–	125.726	–	60188960
35	723.037	–	201.838	–	95656368
40	1,931.644	–	460.450	–	167734448
45	4,413.650	–	1,321.359	–	239649808
50	4,354.936	–	1,402.677	–	271487776

Specification 4

N	Non-MC Time (s)	Model Checking Time (s) CDL*K	CDL*K(OP)	CTL*K	BDD Memory (bytes) CDL*K	CDL*K(OP)	CTL*K
3	0.004	0.069	0.017	3.265	13754080	10247392	90855936
4	0.015	0.087	0.031	OOM	14200800	10583616	OOM
5	0.035	0.244	0.108	–	19087792	12491344	–
6	0.071	0.205	0.115	–	18288224	12516480	–
7	0.154	0.350	0.218	–	29372240	13930960	–
8	0.284	0.572	0.376	–	35499056	15589008	–
9	0.429	0.883	0.658	–	36291680	17283936	–
10	0.815	1.500	1.091	–	46844272	20384432	–
12	1.738	2.415	2.220	–	44009616	36089232	–
15	7.026	6.114	6.009	–	51791104	49690336	–
20	38.035	23.943	23.197	–	58944000	58364864	–
25	98.500	81.433	71.918	–	60745264	58582736	–
30	282.580	323.992	310.658	–	60349744	60089616	–
35	713.533	655.159	555.417	–	95831648	95541680	–
40	1,938.729	1,067.446	1,127.992	–	167973376	167573424	–
45	4,304.172	3,968.253	3,882.683	–	239834128	239410800	–
50	4,399.351	7,866.579	7,618.838	–	271944480	271404256	–

Table 8.5: Experimental results for Specifications 3 and 4 for the Prisoners scenario.

our hypothesis that our implementation for CTL*K and CDL*K is scalable in terms of the size of the model. Consider that we were able to verify all of the specifications over state spaces as large as 2.1923×10^{25} states (which is the case for $N = 50$). We were even able to verify Specification 2 over 1.60202×10^{35} states ($N = 70$).

- The impact of our CDL*K algorithm optimisations (introduced in Section 5.4) varies depending on the specification.
  - For specifications 1 and 4 there is generally a small performance improvement; the relevant optimisation here is the pruning of redundant states from ϵ-NFAs, which reduces the state space by a constant factor. Note that exceptions do exist ($N = 15$ for specification 1, $N = 40$ for specification 4), perhaps because some overhead is necessary to carry out these optimisations.
  - For specification 3, the optimisation that is relevant also appears to be the pruning of redundant states from ϵ-NFAs. However, as the automaton for this formula is substantially more complex than the automata for specifications 1 and 4, the "constant factor savings" in the state space are significantly larger. Efficient conjunct computation is also significant for this specification, owing to complexity of the automaton.
  - For specification 2, although pruning of redundant states does take place, we suspect propositional shortcircuiting is the more significant optimisation, leading to a dramatic performance improvement. The entire expression inside the box modality is only dependent on the current state, and can thus be evaluated first.
- It is perhaps reassuring, if expected, to see the CDL*K algorithm perform better than the CTL*K algorithm on an approximation of specification 4 in CTL*K. Clearly, the CTL*K property is stronger, and it is the weakest strengthening of the original condition that we were able to construct – in that sense, a direct comparison might not be fair. Nonetheless, this example does serve to highlight the richer expressivity of CDL*K.

8.8 Go-Back-N

Go-Back-N is a pipelined network communication protocol, with the aims of providing recovery from dropped packets as well as improving channel utilisation in high-latency conditions [46]. Much like the Bit Transmission Protocol discussed in Section 8.6, we have a Sender which aims to communicate information to a Receiver over an unreliable channel which may drop messages (though it will not corrupt messages). The Sender has a buffer of M data packets which he wishes to send to the Receiver; he transmits these along with a sequence number (which ranges from 1 to M). Furthermore, he does not wait for an acknowledgement after each packet; instead, he keeps transmitting up to a window of N unacknowledged packets (i.e. if the Sender has received an acknowledgement for packet k, he can try to send all packets from $k+1$ to $k+N$). Upon receipt of a packet, the Receiver sends an acknowledgement indicating the last packet it has received *in sequence*. We illustrate a possible run of the protocol in Figure 8.5.

We construct a model of a synchronous variant of the Go-Back-N protocol in ISPL, as follows. An example of our ISPL implementation may be found in Appendix B.

- We use bits for the data packets. This may seem inefficient as we are using $\log M$ bits of overhead for 1 data bit – however, packets in practice are substantially larger than 1 bit. We use bits for verification because this is sufficient for us to assert properties concerning correctness of the protocol (e.g. the Receiver and Sender are never in conflict).

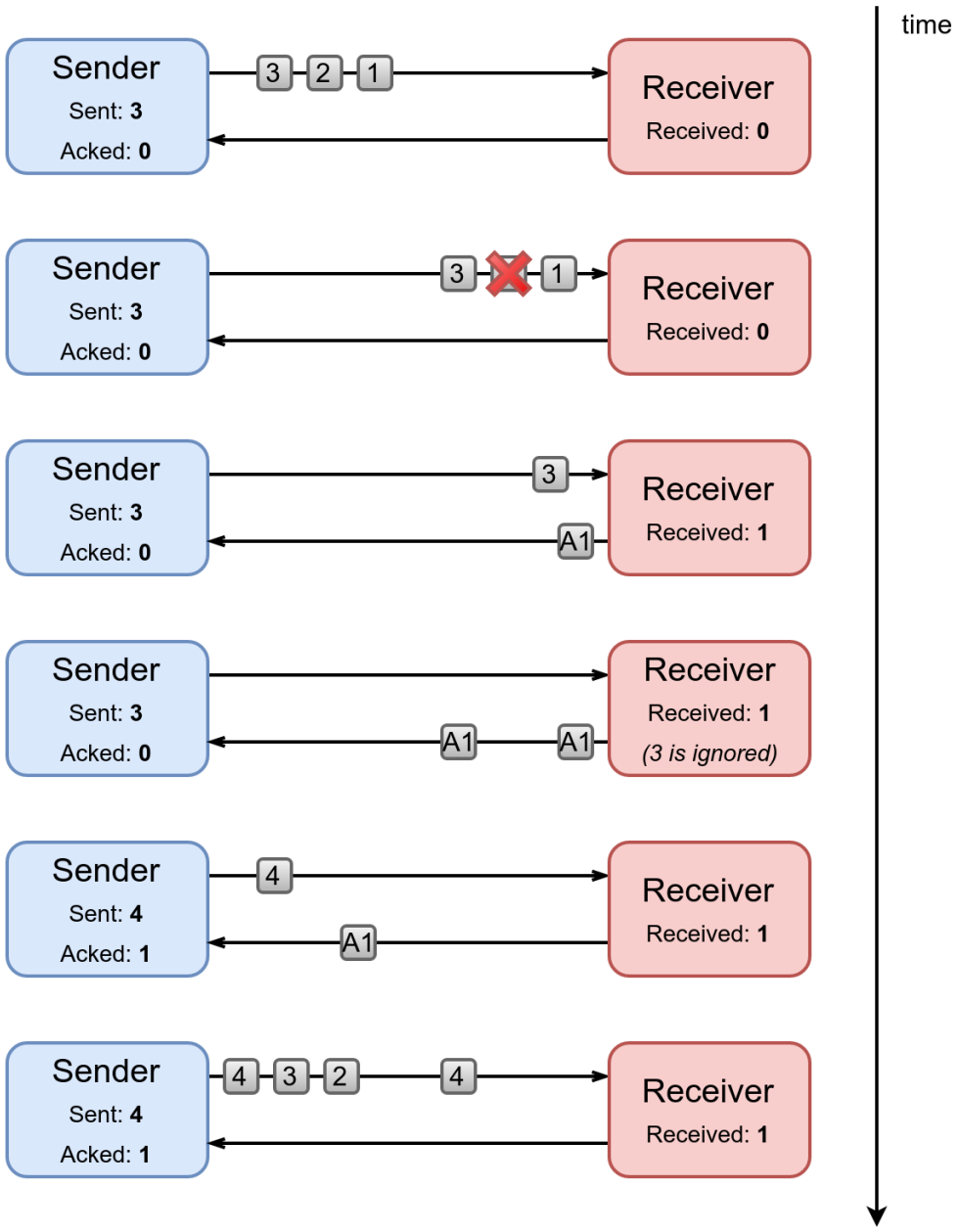


Figure 8.5: Graphical representation of a sample run of the Go-Back-N protocol for $N = 3$. The messages from the Sender to the Receiver are annotated with sequence numbers; the messages from the Receiver to the Sender indicate acknowledgment of the corresponding packets. Packet 2 is lost in transmission on the first run (step 2); however, because the Receiver indicates that it has missed a packet when it receives packet 3 (step 4), the Sender knows it needs to retransmit packet 2 when it receives the duplicate ACK (step 6).

- The `Environment` is used to model a three-step bidirectional communication channel; a packet that the Sender places on the channel, if successfully delivered, will be visible to the Receiver three rounds later. The channel may drop packets in step 1 or step 2 in either direction; the `Environment`'s actions involve choosing which packets (if any) are dropped.
- The `Sender` agent has M bits to transmit to the receiver, as well as two counters *recack* and *dispatch* which, respectively, keep track of the last acknowledgment received and the last packet sent. It follows the protocol described above; in the event it receives a duplicated ACK or does not receive any relevant ACKs before it reaches the end of its transmission window, it restarts from just after the last packet that was ACKed.
- The `Receiver` agent receives the M bits that the Sender sends; after receiving the bit i, it acknowledges this by sending an ACK message i to the Sender. In the event that the receiver receives a bit out of order (e.g. it receives a bit with sequence number 2 from the start), it ignores the bit and repeats the current ACK message, if applicable[4].
- We define as atomic propositions:
  - $mismatch_i$, which holds if the Receiver has "received" a bit in position i that is different from that of the Sender;
  - *envworks*, which holds if the channel is delivering messages in both directions (for one round);
  - *envbroken*, which holds if the channel is dropping messages in both directions (for one round);
  - $rbit_i = v$ ($i \in \{1, \ldots, M\}$, $v \in \{0, 1\}$) which holds if the receiver's view of bit i is v.

Unfortunately, the state machine grows complex fairly quickly, even for small values of N and M [46]. Nonetheless, we attempted to verify several properties:

1. **The Sender and Receiver are never in conflict (on the bits the Receiver knows).** This is true, as the system does not corrupt bits. We can write this in LDL or LDL_f as

$$[\top^*]\left(\bigwedge_{i=1}^{M}(\neg mismatch_i)\right)$$

   (Of course, this property can be written similarly in LTL or CTL; that said, the focus of this example is more to compare the performance of finite and infinite trace semantics.)

2. **If the channel always works, then eventually everyone knows that the Receiver knows the value of the last bit.** This is true over infinite trace semantics, but *not* over finite trace semantics (since we can terminate a trace early). We can express this in LDLK or LDL_fK, where Γ refers to the group of all of the agents, as

$$([\top^*](envworks)) \rightarrow \langle\top^*\rangle E_\Gamma\left(K_R(rbit_M = 0) \vee K_R(rbit_M = 1)\right)$$

3. **If the channel only works for bursts of one or two steps at a time, but quickly recovers when it fails, then it is possible for the protocol to succeed – that is, eventually everybody knows that the Receiver knows the value of the last bit.** (Here, by *quickly recovers*, we mean that the channel is only down for one step at a time.)

[4]This is an inefficiency of Go-Back-N that is addressed by the *selective repeat* protocol [46], though the state machine becomes even more complex.

This is sufficient, as enough messages will be delivered since the Sender keeps sending a steady flow of messages. It can be expressed in CDL*K or CDL*$_f$K as

$$envworks \rightarrow E(\langle((envworks; envbroken) + (envworks; envworks; envbroken))^*\rangle\psi)$$

where ψ refers to the protocol succeeding – that is,

$$\psi = E_\Gamma(K_R(rbit_M = 0) \vee K_R(rbit_M = 1))$$

Note that if we changed this property to *common knowledge*, it would not hold (and would not hold even if the channel always worked); the Receiver does not receive ACKs for the ACKs he sends, so the Receiver has *no way* of knowing that the Sender knows he knows the value of the last bit.

Notice that for this scenario we need to document the non-model checking time for infinite and finite trace semantics separately, because under finite traces we double the size of the model by adding the path terminator agent, and also add a (small) additional preprocessing and parsing load. Our results are presented in Table 8.6. From the data, we draw the following conclusions:

- Generally, as M increases, the time required to build the model and verify specifications, as well as the BDD memory usage all increase. This is unsurprising, since the size of the (reachable) state space increases.
- The pattern is not as clear when N increases; as we increase N from 2 to 3 we obtain increases in the state space, but this is not necessarily the case when we increase N from 3 to 4; our experimental results include both increasing and decreasing reachable state space sizes, as well as times for both building the model and model checking itself.
- Furthermore, observe that when $M = 3$ and we increase N from 3 to 4, the results for both model checking and non-model checking times, as well as BDD memory used are very similar (and the reachable state space also has the same size). This is in line with expectations, since the semantics are the same in both instances; the Sender is allowed to send all of the bits without any acknowledgements.
- Also, generally verifying properties using our algorithm for finite traces adds a cost in terms of non-model checking time.
  - This is in line with expectations, since as explained earlier the size of the model is doubled, and we also need to preprocess the input ISPL file. We found that the number of reachable states in the finite traces case was consistently just under double that in the infinite traces case. Note that it was not exactly double, because for initial states in the original model that are never revisited, the combined state of said initial state and the path terminator being *dead* is not reachable in the new model.
  - There is a notable exception for $M = 4$ and $N = 3$ that was consistent across the specifications; again, we attribute this to empirical efficiency of BDDs.
- In terms of model checking time, verifying properties over finite traces is typically more costly.
  - This is unsurprising. Recall that to determine whether paths that satisfy our specification hold in our model, we need to check $EG\top$ with fairness constraints in the composition of the model and our symbolic Büchi automata for the formulae. This step is linear in the size of the model and the number of fairness constraints; our

Specification 1

		Non-MC Time (s)		Model Checking Time (s)		BDD Memory (bytes)	
N	M	LDLK(OP)	LDL$_f$K(OP)	LDLK(OP)	LDL$_f$K(OP)	LDLK(OP)	LDL$_f$K(OP)
	3	0.555	0.876	0.035	0.071	13935168	16992336
	4	1.852	2.733	0.089	0.333	31431008	42200304
2	5	7.882	26.434	0.230	0.534	38859136	45837552
	6	21.470	32.453	0.513	0.862	55756992	57809488
	7	57.575	78.479	3.020	9.310	65904992	87732560
	3	1.668	2.825	0.106	0.216	31742272	32853616
	4	6.023	5.254	0.227	0.408	47626528	48179184
3	5	43.049	54.116	1.033	3.486	46593184	64310640
	6	146.583	149.138	8.159	27.833	111751232	150102160
	7	622.082	1,344.836	65.719	332.687	282844416	375338960
	3	1.673	3.131	0.106	0.216	31742272	32853616
	4	12.681	19.723	0.583	1.349	53341856	58823408
4	5	36.367	48.294	1.127	8.044	58447680	90168944
	6	96.940	133.701	7.386	30.001	115645504	155371344
	7	506.742	813.535	25.191	191.873	269264992	378968752

Specification 2

		Non-MC Time (s)		Model Checking Time (s)		BDD Memory (bytes)	
N	M	LDLK(OP)	LDL$_f$K(OP)	LDLK(OP)	LDL$_f$K(OP)	LDLK(OP)	LDL$_f$K(OP)
	3	0.554	0.868	0.242	0.299	31676352	41972784
	4	1.852	2.663	0.797	0.864	51096064	56373776
2	5	7.840	24.829	1.864	7.283	59379616	63214768
	6	20.840	33.412	8.083	9.120	57835488	82117552
	7	56.313	79.921	253.259	40.706	138874272	191735568
	3	1.672	3.180	0.975	1.493	53353280	58174352
	4	6.060	5.035	2.004	3.520	57011904	62035504
3	5	39.424	56.335	30.385	23.420	67000544	130813648
	6	139.527	148.035	228.246	68.292	156081952	254456368
	7	622.950	1,322.311	2,331.437	606.917	410939520	903662864
	3	1.671	2.951	0.981	1.403	53353280	58174352
	4	12.777	17.343	15.226	13.585	62455008	80508048
4	5	27.266	46.589	62.521	35.942	68901280	178513808
	6	112.874	134.280	223.339	79.966	155536160	249179600
	7	502.012	828.060	2,428.861	750.290	716820352	707239696

Specification 3

		Non-MC Time (s)		Model Checking Time (s)		BDD Memory (bytes)	
N	M	CDL*K(OP)	CDL*$_f$K(OP)	CDL*K(OP)	CDL*$_f$K(OP)	CDL*K(OP)	CDL*$_f$K(OP)
	3	0.553	0.872	0.301	0.654	37254208	44368880
	4	1.831	2.597	0.868	2.534	51022752	60351184
2	5	7.782	21.638	2.584	48.234	54868128	61721168
	6	21.064	27.076	8.350	82.197	59992096	66953136
	7	56.144	67.026	221.307	587.844	138291104	195161360
	3	1.658	2.918	1.056	4.382	57462208	59780464
	4	6.013	4.650	2.447	7.653	57940896	58999216
3	5	40.948	50.262	26.029	165.710	68177024	135566928
	6	131.702	148.430	133.488	414.669	150692640	222690960
	7	613.147	1,283.107	1,567.676	4,509.025	388678912	749100848
	3	1.659	2.927	1.052	4.486	57462208	59780464
	4	10.224	18.685	8.054	51.139	60842528	69763824
4	5	37.677	44.169	34.074	227.603	66147104	179391344
	6	97.237	120.720	124.864	410.961	151523552	225101712
	7	501.267	741.615	1,159.449	4,967.685	761274912	604519376

Table 8.6: Experimental results for the Go-Back-N scenario.

reduction doubles the size of the model (by introducing the path terminator) and also adds one fairness constraint ($\neg Alive$). Furthermore, the finite translation increases the size of the formula being verified (it adds a propositional operator and an instance of the *Alive* proposition for each modality), potentially leading to a further increase in the time required.

  - However, this is not the case for Specification 2; this is because the specification holds over infinite traces, and requires us to iterate through the full depth of the model (since for the proposition to hold we need the Sender to receive an ack for bit M). Conversely, for finite traces we can very quickly show that the proposition does not hold, since the trace can be aborted early. We profiled several runs of our tool with `callgrind`, and found that fewer preimages were required for finite traces (e.g. for $M = 4$ and $N = 2$, we required 99 preimages to verify the property over infinite traces, but only 21 to show that it did not hold over finite traces – even though each preimage in the infinite trace case is costlier owing to the larger model). Furthermore, as M increases, the number of preimages required for infinite traces increases[5], while that over finite traces appeared to hold constant (21 in all cases).

- Over infinite traces, verification of Specification 1 scales most easily, followed by Specification 3, with Specification 2 being slowest.

  - This may be surprising given the size of the formulae, as the order might seem reversed from that (i.e. Specification 2 is smallest and Specification 1 is largest). However, consider that for Specification 1, the propositional shortcircuiting optimisation allows us to treat the entire expression within the box modality as a single atomic proposition when verifying it. Specification 3 may seem large, but in terms of automata construction we only need to build the automata for the expression within the E-quantifier; after we find the states in which the E-quantified subformula holds, we also use propositional shortcircuiting to return the final result.
  - Over finite traces, verification of Specification 2 seems to scale better than that of Specification 3, perhaps owing to the relative ease of showing Specification 2 is *not* satisfied over finite traces as discussed above.

- The propositional shortcircuiting optimisation also applies to verifying Specification 1 over finite traces. This is suggested by the memory usage for this specification, which remains on the same order of magnitude as that for Specification 1 over infinite traces. (Without the optimisation, the memory usage will tend to increase substantially; consider the results for Specification 2 of the Prisoners scenario, which is similar, in table 8.4.) This is true for Specification 3 as well, though the effect may not be as pronounced.

8.9 Summary

In general, we observe that our extensions of MCMAS to support the verification of properties in Ltlk, Ctl*k, Ldlk and Cdl*k, along with the finite trace extensions of the last two, generally have lower performance than that of MCMAS on Ctlk specifications. This is unsurprising given the higher theoretical complexities of model checking our additional specification languages (all PSPACE-complete, as opposed to Ctlk model checking which is in P).

The degree of this slowdown depends on the specification being verified. Generally for small

[5] We needed 75 preimages for $M = 3, N = 2$ and 123 for $M = 5, N = 2$; we conjecture $24M + 3$ overall.

specifications and/or specifications with substantial non-temporal components, our extensions while slower are still on the same order of magnitude as the original CTLK algorithm, as far as model checking time is concerned. This is in line with expectations, as while the model checking problems for all of our additional specification languages are PSPACE-complete, they are *fixed parameter tractable*. In particular, the runtimes for our model checking algorithms are linear in the size of the model, and exponential only in the size of the formula. For example, consider Specification 2 for the Prisoners scenario (Section 8.7); we were able to verify the protocol's safety in CTL*K and CDL*K even when $N = 70$ (yielding a state space with more than 10^{35} reachable states).

The main bottleneck as far as LTLK and CTL*K model checking performance are concerned especially for large formulae appears to be the checking of $EG\top$ with fairness constraints; this makes sense as the additional fairness constraints introduced can add a substantial overhead to this check. In the interest of time, we did not focus on optimising this algorithm, so there may have been other inefficiencies that we did not consider. On the other hand, for LDLK and CDL*K, the main performance bottlenecks appear to be performing the symbolic breakpoint construction and checking of $EG\top$ with the model of our system, which is in line with our expectations. We have taken steps to address the former through several performance optimisations (Section 5.4) and shown that these optimisations can be useful in practice. Unfortunately, the latter appears inevitable especially in models for which counterexamples and/or witnesses are necessarily long (and thus require many iterations before convergence).

Of course, there are reasons besides performance for which our extensions are useful. They contribute significantly to the range of specifications that MCMAS can verify; there are many temporal properties expressible in our additional specification languages that are not expressible in CTLK[6]. For example, these include specifications that are concerned with

- properties of individual paths (e.g. Specification 2 for Bit Transmission and Go-Back-N),
- parity (e.g. Specifications 1 to 3 and 5 for the Counter, Specification 3 for the Prisoners),
- arbitrary-length alternating sequences (e.g. Specification 4 for the Counter), and
- more complex regular properties over individual paths (e.g. Specification 4 for Prisoners, Specification 3 for Go-Back-N).

One can also consider the ten properties we used to compare the expressivity of various temporal logics in Section 2.3; MCMAS would previously only have been able to verify the properties expressible in CTLK (i.e. statements 1–4, 8 and 9), while our extensions allow verification of all ten properties (since all of the properties are expressible in CDL*K, shown in Section 6.1).

Furthermore, we extended MCMAS's capabilities to verify properties over finite trace semantics, as introduced in Chapter 7. This is novel and different from the other specification languages (as well as the existing specification languages supported by MCMAS), which are verified over infinite traces.

In that sense, our extensions are not designed to replace the original tool; they are designed to add additional expressive power to MCMAS (even if the ability to verify some of these properties comes with a high cost). We find that in general, they are able to handle reasonably large state spaces, especially if specification formulae are not excessively large.

[6]MCMAS also supports ATLK, though this is not relevant as our extensions were not concerned with verifying strategic ability.

Chapter 9

Project Evaluation

In this chapter we begin by evaluating the strengths and weaknesses of our theoretical contributions. We then evaluate the strengths and weaknesses of our implementation of these algorithms in MCMAS-Dynamic.

9.1 Theory

Our project focused on developing practical, efficient model checking algorithms for LTLK, CTL*K and LDLK. We also explored interesting temporal and epistemic extensions of LDLK (CDL*K, as well as finite trace semantics), formalised them and developed model checking algorithms for them as well. Our theoretical contributions are as follows:

1. **Epistemic Modalities for LTLK and CTL*K**. We presented an algorithm for handling epistemic modalities within LTL and CTL*, using a recursive descent approach. We showed that these algorithms preserve the theoretical complexities of the relevant model checking problems – that is, $O(2^{|\phi|} \times |\mathcal{I}|)$, if one uses an optimal LTL algorithm.

2. **Formalism of CDL*K**. We added epistemic modalities to LDL, giving rise to LDLK, and formalised its full-branching time extension CDL*K. We showed that the LDLK and CDL*K model checking problems are both PSPACE-complete.

3. **Practical model checking algorithms for LDLK and CDL*K**. We modified the construction of [41] for LDL by introducing the notion of *critical sets*. This allowed us to concretely construct alternating automata accepting precisely the models corresponding to a given LDL formula in a bounded amount of time. We proved that our construction is correct, and that it runs in $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$ time. We generalised our algorithm to handle epistemic modalities, as well as our full branching time extension, while preserving the same asymptotic time complexity. We also proposed several optimisations that simplified the alternating automata generated by this procedure, and proved their correctness.

4. **Formalism of CDL*K over finite traces.** We formalised the full-branching time extension of LDL_f with epistemic modalities, $\text{CDL}^*{}_f\text{K}$. We showed that $\text{CDL}^*{}_f\text{K}$ model checking is also PSPACE-complete.

5. **Algorithm for CDL*K and LDLK over finite traces.** We presented a reduction of $\text{CDL}^*{}_f\text{K}$ model checking to CDL*K model checking, and justified its correctness. Since $\text{CDL}^*{}_f\text{K}$ subsumes LDL_fK, the algorithm is also applicable to LDL_fK properties.

9.1.1 Strengths

We observe the following strengths in our theoretical contributions:

1. **Optimal time complexity for LTLK and CTL*K.** Our algorithms proposed for LTLK and CTL*K can achieve optimal asymptotic time complexity, provided the underlying LTL model checking algorithm has optimal asymptotic time complexity[1].

2. **Amenability to practical symbolic implementations.** Our novel algorithms for LDLK and CDL*K are largely amenable to symbolic model checking. An exception might be the states of the alternating automaton construction (Section 5.1.1), but we believe this is not an issue as its size is linear in the specification size. The transitions of the alternating automata can be computed symbolically.

3. **Balance between expressive power and complexity.** We put forth the formalism of CDL*K, which has substantial expressive power in that it can express all ω-regular properties, and subsumes many popular temporal-epistemic logics (such as LTLK, LDLK, CTLK and CTL*K). However, CDL*K remains in the PSPACE complexity class; in theory, model checking CDL*K is no harder than model checking LTL.

4. **Formal correctness.** We have formalised the concepts used in our model checking algorithms; furthermore, we either reuse standard, established techniques (e.g. symbolic breakpoint construction, Section 2.6.4) or use original techniques for which we have proved correctness (e.g. non-critical set path elimination, Theorem 5.1 in Section 5.1.2). Hence, we can be confident that our model checking algorithms are formally correct.

9.1.2 Weaknesses

On the other hand, we also observe two main weaknesses of our theoretical contributions:

1. **Uncertainty concerning optimality of our LDLK model checking algorithm.** We have provided an $O(\max(2^{|\phi| \log |\phi|}, 3^{|\phi|}|\mathcal{I}|))$ bound on the runtime of our LDL model checking algorithm. It is uncertain whether a tighter bound is possible (we suspect a lower bound is $O(2^{|\phi|}|\mathcal{I}|)$, since the LTL model checking problem reduces to the LDLK model checking problem). Nonetheless, our algorithm is singly exponential with exponentiality only in formula size, which we believe renders it suitable for many practical applications.

2. **Suboptimal space complexity.** Our algorithms for model checking LDLK and CDL*K (as well as for these languages over finite trace semantics) involve the symbolic breakpoint construction, which constructs a Büchi automaton with $O(3^{|\phi|})$ states. Since we eagerly construct said automaton, our algorithms may require exponential space – this is suboptimal, since the model checking problem for these languages is in PSPACE. We believe that the empirical efficiency of binary decision diagrams compensates for this in practice.

9.2 Implementation

Our project also focused on implementing the model checking algorithms developed for each specification language, and evaluating the correctness, performance and scalability of our implementation. In addition to the three languages we originally targeted (LTLK, CTL*K and

[1] It is worth noting that the algorithms we actually implemented eagerly construct the Büchi automata and thus are not asymptotically optimal; however, we believe this is outweighed in practice by the average-case efficiency of symbolic model checking.

LDLK), we also did this for CDL*K, and the finite trace semantics versions of both LDLK and CDL*K. Our practical contributions were as follows:

1. **MCMAS-Dynamic**. We extended the MCMAS model checker with support for each of the aforementioned specification languages. For LTLK and CTL*K, this involved a symbolic implementation of the tableau construction of [28], as well as of the algorithms we introduced; for LDLK and CDL*K, this involved a *hybrid* implementation of our alternating automaton construction (explicit states, but symbolic transitions), followed by a *symbolic* implementation of the breakpoint construction. For finite trace semantics, we implemented our reduction of the finite trace model checking problem to an equivalent model checking problem over infinite traces. We also extended the MCMAS parser to support specifications in each of the newly supported languages, as well as implemented counterexample generation for LDLK or universally quantified CDL*K formulae (and witness generation for existentially quantified CDL*K formulae).

   Furthermore, we implemented all of the automata simplification optimisations for LDLK proposed in Section 5.4.3, as well as investigated and optimised the sequencing of BDD operations for computing the symbolic breakpoint construction (Section 5.4.4). Since our CDL*K and finite traces algorithms all use our LDLK model checking procedure as a subroutine, they also benefit from these optimisations.

2. **Experimental evaluation.** We evaluated how our implementation scaled on several scenarios, both in terms of models as well as formulae of increasing size. We compared our extension's performance against MCMAS on equivalent CTLK formulae, as well as against itself (e.g. on equivalent LTLK and LDLK formulae). We also demonstrated the rich expressivity of the additional specification languages supported, and considered and analysed the impact of our performance optimisations.

9.2.1 Strengths

We observed the following strengths in our tool as well as relevant implementation work:

1. **First model checker for LDL and its extensions.** To the best of our knowledge, no other model checking tool supports LDL specifications. Our extension of MCMAS adds support for not just LDL or its epistemic extension LDLK, but also for the various extensions we have explored (full branching time semantics, finite trace semantics). We believe this novelty is the greatest strength of our project.

2. **Largely symbolic implementation.** We have represented all additional automata constructions as well as model checking steps symbolically, apart from the states of the LDL alternating automaton construction. We believe this is acceptable, as these alternating automata are linear in the size of the formulae being verified [41]; transitions (which may be exponential in the formula size) are represented symbolically, and the construction of the nondeterministic Büchi automaton we use to check the property (which may incur an exponential blow-up) is also implemented symbolically. Our experimental evaluation (Chapter 8) verifies that in many cases, CDL*K model checking can be practical.

3. **Expressive power.** Our tool supports specifications in CDL*K, which as shown in Section 6.1 is a very expressive temporal-epistemic logic. At the path level, CDL*K allows all ω-regular properties to be expressed; at the state level, CDL*K supports mixed quantification over paths. Furthermore, we also support the finite trace variant of CDL*K.

This significantly enhances the expressivity of our tool, compared to "vanilla" MCMAS (CTLK and ATLK), MCK (CTL*K) or NuSMV (CTL and LTL). Yet, as discussed above, our implementation of CDL*K model checking is still practical in many cases.

4. **Comprehensive experimental evaluation.** Our evaluation covered how our tool scales over several pertinent parameters (formula size, model size, optimisations, finite vs. infinite traces). We were able to demonstrate our tool's relative resilience to large models (best exemplified through the Prisoners scenario, Section 8.7) and its high expressive power (summarised in Section 8.9), as well as the effectiveness of our optimisations. We were also able to demonstrate its worst-case $O(2^{|\phi| \log |\phi|})$ behaviour (Specification 2 in the Bit Transmission scenario, Section 8.6).

9.2.2 Weaknesses

Conversely, we also observed several potential areas for improvement:

1. **Low performance.** For equivalent specifications, we observe that our extensions tended to be substantially slower than the existing CTLK implementation in MCMAS. This is in line with expectations, given the higher computational complexity of the model checking problems for the various languages we considered (all PSPACE-complete, as opposed to P for CTLK).

   We believe that our extension is practically useful, nonetheless, as it supports a wide variety of properties not expressible in CDLK. Furthermore, our algorithms are *fixed parameter tractable* – even though they may require exponential time, we have exponentiality *only* in the size of the formula. Thus, as shown in Chapter 8, we were able to handle reasonably large (or, in some cases, very large) state spaces if formulae were small, which has been claimed to be the case in practice [79]. In several cases, our LDLK optimisations were useful in speeding up verification of LDLK and CDL*K properties to require time on a similar order of magnitude as that for CTLK, though this is unlikely to be possible in general[2].

2. **Code health.** We attempted to maintain a relatively clean design as far as our extensions are concerned (for example, introducing several abstractions in the form of automata, modalities, nodes etc., as in Section 5.3.1). However, in the interest of time we decided to follow many of the existing code patterns in MCMAS, even if they were not optimal from a software engineering point of view (for example, we used the existing monolithic `bdd_parameters` struct, as well as depend on several global variables such as `agents`).

   This also made unit testing difficult, as discussed in Section 8.2.1. We mitigated this by implementing a framework for automated system testing, and creating a large set of test specifications which cover various formulae across our specification languages (again, discussed in Section 8.2.1). We also checked that our tool was consistent across equivalent formulae in multiple specification languages (Section 8.2.2). However, our tests were still limited in that they did not cover counterexample generation (owing to nondeterminism) and were at a relatively coarse granularity. We nonetheless found said tests invaluable in giving us confidence that our changes were correct.

[2] We say *unlikely* because it is not definitively known whether P is a strict subset of PSPACE, so it could be the case that these model checking problems actually have the same theoretical complexity.

Chapter 10

Conclusions

As discussed at the beginning of Chapter 5, ω-regular properties are useful in systems verification; yet, we were not able to find model checkers that support automatic verification of all such properties. We addressed this problem by devising a novel adaptation of the alternating automaton construction from [41] using critical sets, which admits a concrete implementation, and used this to develop the first model checker for LDLK.

We also considered extensions of LDLK such as CDL*K and finite trace semantics that further improve the expressive power of LDLK, and developed algorithms and tool support for these extensions. Although the theoretical complexity of verifying such properties can be high, our experimental results show that in many scenarios, these techniques are still feasible in practice and can be used to verify agents' behaviour, as well as identify counterexample or witness traces where appropriate (which, in turn, may be useful in identifying and fixing bugs).

10.1 Summary of Work

We first summarise the work that we have done, considering this against our objectives set out in Section 1.1. We laid out our objectives in an iterative fashion, seeking to develop, implement and then evaluate algorithms for temporal-epistemic logics of increasing expressive power, as laid out in Figure 1.1.

We began with Linear Temporal Epistemic Logic (LTLK). We reduced the LTLK model checking problem to the standard (that is, non-epistemic) LTL model checking problem. We showed that our algorithm had runtime exponential in the formula size but linear in the model size. We then concretely implemented the tableau construction with some adaptation, owing to the peculiarities of MCMAS, and thus added support for symbolic model checking of LTL and LTLK properties to MCMAS (inclusive of counterexample generation). We also uncovered and fixed a subtle bug concerning MCMAS's existing counterexample generation implementation.

We then extended our work to Full Branching Time Epistemic Logic (CTL*K), using the recursive descent ideas put forth in [39]. As before, we implemented support for symbolic model checking of CTL* and CTL*K properties to MCMAS (again, inclusive of counterexample generation).

Next, we tackled the problem of model checking Linear Dynamic Epistemic Logic (LDLK) specifications. We considered the interaction of LDL with epistemic modalities, and showed that LDLK is PSPACE-complete. We developed a novel construction for LDL which helps us

eliminate the requirement of considering every ϵ-path from the construction in [41]. We showed that the alternative construction is correct, and runs in time singly exponential in the size of the formula but linear in the size of the model. We implemented the first model checker for LDL and LDLK (to the best of our knowledge), using our alternating automaton construction and the symbolic breakpoint construction of [20]. We also investigated multiple approaches for improving the performance of our implementation, both in terms of simplifying automata as well as making efficient use of BDDs. We showed that these optimisations were correct, and implemented all of them in our tool. We also implemented support for counterexample generation.

We then introduced the full branching time extension of LDLK, CDL*K, which adds support for existential path quantification to LDLK. We formalised its syntax and semantics, and provided a model checking algorithm that used our LDLK algorithm that was singly exponential in the size of the formula and linear in the size of the model. We used this algorithm to show that CDL*K is PSPACE-complete, which gives it the same theoretical complexity as LTL even though it is substantially more expressive (we showed it subsumed CTL*K and LDLK, which themselves subsume LTLK). We also used this algorithm to implement the first model checker for CDL* and CDL*K (again, to the best of our knowledge).

We then considered LDL_f, which is LDL over finite traces, and introduced our novel extension $\text{CDL*}_f\text{K}$ (which is CDL*K over finite traces). We provided a model checking algorithm for $\text{CDL*}_f\text{K}$, showed its correctness, and also showed that this logic is PSPACE-complete with runtime exponential only in the size of the formula. We extended our tool to allow verification of $\text{CDL*}_f\text{K}$ properties, including counterexample generation.

We then tested the performance of our model checker over a wide variety of scalable scenarios and specifications. We showed that while the worst-case $O(2^{|\phi| \log |\phi|})$ behaviour can indeed occur, for reasonably small specifications the tool is still able to verify properties over large state spaces (possibly as large as 10^{35} states in some cases). We also put forth many examples of properties that our extension can handle, that could not previously be verified by MCMAS owing to the limited expressivity of being restricted to CTLK alone.

In terms of functionality, we have been largely successful in our goals. We have indeed developed algorithms for model checking each specification language, justified their correctness, implemented model checkers for each language and subjected our implementations to significant acceptance and scale testing. There is still certainly room for improvement, especially in terms of performance. To some extent, this is an unavoidable consequence as our specification languages have substantially higher model checking complexities. We have mitigated this through several performance optimisations, and believe the additional expressivity of our tool certainly compensates for said low performance. There also remains much scope to push this further, as outlined in Section 10.2.2.

10.2 Future Extensions

We believe we have made significant progress towards practical LDLK and CDL*K model checking, by implementing the first LDL model checker, as well as implementing support for several temporal and epistemic extensions. Nonetheless, there is much room for this project to be extended, both in terms of theory and implementation. We partition this section into a discussion of extensions concerning expressive power and/or functionality of our tool or logics, as well as extensions concerning performance or reliability of our algorithms or implementations.

10.2.1 Expressivity and Functionality

- **Metric LDL or CDL*.** The *metric* extension of LDL or CDL* includes modalities which have a constant bounded match length; for example, $[p; q^*]_{\leq 4} r$ holds iff after every match of $p; q^*$ *of length at most* 4, r holds. Currently, we can simulate this by enumerating all possible matches (e.g. $[(p+p; (q+q; (q+q; q))]r$) but this needs to be done manually, and tends to be infeasible in practice especially for large time bounds. A first improvement could be to add a simultaneous transition into a counting automaton that keeps track of the time left before one has to exit the nodes in the ϵ-NFA; alternatively, one may attempt to extend the breakpoint construction to account for this (as suggested in [41]).

- **Parametric LDL or CDL*.** The *parametric* extension builds on the above metric extension, by allowing users to specify arbitrary bounds on matches (and having the tool automatically determine the optimal bound values). For example, given $\langle p; q^* \rangle_{\leq x} r$, the tool should either indicate that the formula is unsatisfiable for any value of x, or indicate the smallest value of x for which it holds. More detail can be found in [41].

- **Heuristics for shorter witnesses/counterexamples.** We currently rely on the algorithm implemented to determine witnesses for EG in MCMAS to determine counterexamples or witness traces for LTLK, CTL*K, LDLK and CDL*K formulae (including the finite trace variants of the latter two)[1], which is based on the algorithm from [29]. Shorter counterexamples are often easier to understand and debug [85], and there exist methods such as that presented in [49] which, while potentially slower than that proposed in [29], have stronger guarantees about the lengths of counterexamples produced. It may be worth noting that the problem is NP-complete in general [29].

- **LDLK or CDL*K interpreted over probabilistic systems.** Interpreted systems do not currently have a notion of probability (only nondeterminstic choice). Currently, logics like CDL*K only allow us to quantify universally or existentially over paths. It is thus not possible to specify properties concerning probability (for example, "the probability is less than 10^{-8} that the trains will crash" as opposed to "it is not possible for the trains to crash"). Adding support for such properties could be done by extending interpreted systems with action probabilities (effectively, defining a discrete-time Markov chain over global states) and extending specification formulae to support reasoning about probability.

  The PRISM model checker [59] supports a logic which subsumes PCTL*, an extension of CTL* with probabilistic operators, though it currently does not directly support epistemic modalities. Adding support for PCTL*K (or probabilistic LDLK or CDL*K) might allow verification of interesting properties, such as safety of multi-party computation protocols. For example, using the dining cryptographers scenario (Section 8.4), this allows us to verify an even stronger property than Specification 1; we can verify that if the coin flip was odd and Cryptographer 1 did not pay, then he knows someone paid, but does not know that any other individual cryptographer paid with probability more or less than $\frac{1}{N-1}$[2].

10.2.2 Performance and Reliability

- **More sophisticated automaton minimisation.** We pruned states which did not have incoming non-ϵ transitions from the ϵ-NFAs when constructing the automata for LDL

[1] Though we did fix a bug in it, as discussed in Section 3.2.5.

[2] A protocol which reveals that Cryptographer 2, say, paid with probability $1 - 10^{-5000}$ would still satisfy the specifications in Section 8.4.

(Section 5.4.3). However, it may be possible to achieve even smaller alternating automata using techniques such as *quotientiation*, which involves merging states which are "equivalent" in some sense (see [43] for more detail). This could lead to a significant reduction in the state space of the Büchi automata after we perform the symbolic breakpoint construction. Alternatively, the techniques of [87] may be useful in reducing the size of the Büchi automata, for both the LTL tableau as well as the automata for LDL.

- **Alternative symbolic encodings for LTL or CTL*.** We use the tableau construction of [28], which is also used in other popular model checkers such as MCK and NuSMV. However, approaches that use multiple encodings in parallel have shown encouraging results in practice [80]; this may be useful in boosting the performance of our LTLK and CTL*K implementations.

- **Symbolic breakpoint construction optimisation.** We have implemented a symbolic version of the Miyano-Hayashi construction from [20]. However, the writers also suggest an alternative formulation which is claimed to be potentially more efficient as it exploits parts of the alternating automata that are *linear weak* (that is, parts which have no cycles other than self-loops). Furthermore, the automata we construct are *weak* [41] (that is, each strongly connected component has only accepting or rejecting states); our current approach does not take advantage of this at all.

- **Alternative verification techniques.** We have focused on symbolic model checking (SMC) of LTL, LDL and their extensions, through the use of binary decision diagrams. However, it may be worthwhile investigating the performance of LDL model checking using alternative techniques, such as *bounded model checking* (BMC) – this has also been found to be useful in practice, especially in cases where properties can quickly be shown to be false [18]. BMC is also useful in generating minimal counterexamples [55], which our current SMC-based approach does not guarantee. Furthermore, even within SMC, we can consider the impact of using alternative data structures such as sentential decision diagrams [34], which have seen applications in model checking [67].

- **Investigation of BDD efficiency.** Throughout this project, apart from the efficient conjunct computation optimisation (Section 5.4.4) we have largely treated CUDD, the BDD library that MCMAS uses, as a black box. It may be possible to achieve better performance by optimising the way in which we use the tool (e.g. variable ordering or reordering heuristics, further optimisation as far as order of operations is concerned, tuning of caching parameters etc.). Alternatively, it may be possible to obtain better performance by using different BDD packages, such as BuDDy [2].

- **Parallel construction or verification.** We attempted to carry out the symbolic breakpoint construction in parallel, though this did not work well owing to overheads from CUDD as well as the high cost involved in combining partial results (see the end of Section 5.4.4 for a discussion). Using BDD packages that support concurrency, such as BDDNOW [72] or Sylvan [91] could make this more efficient.

- **Random differential testing.** We drew on the idea of differential testing (Section 8.2.2) to help give us confidence that our tool behaved correctly. A natural extension to this would be to randomly generate formulae in CTLK or LTLK, and then automatically translate them to relevant formulae in the logics that subsume them (CTL*K, LDLK and CDL*K) – a mismatch would indicate a bug in at least one of the algorithms concerned. This would allow us to test our implementations on far more difficult and complex formulae than we were able to manually construct and verify.

Bibliography

[1] "Apache CloudStack: Open Source Cloud Computing". 8 May 2016. <https://cloudstack.apache.org/>

[2] "BuDDy: A BDD package." 6 Jun 2016. <http://buddy.sourceforge.net/manual/main.html>

[3] "The cudd package (Internal). Internal data structures of the CUDD package." 12 May 2016. <http://www.async.ece.utah.edu/~myers/nobackup/ee5740_98/cudd/cuddAllDet.html >

[4] "FindBugs™ – Find Bugs in Java Programs". 10 Jan 2016. <http://findbugs.sourceforge.net/>

[5] "MCK". 10 Jan 2016. <http://cgi.cse.unsw.edu.au/~mck/pmck/>

[6] "MCK 1.1.0: User Manual." 30 May 2016. <cgi.cse.unsw.edu.au/~mck/pmck/mcks/docDownload/manual>

[7] "MCMAS v1.2.2: User Manual". 18 Jan 2016. <http://vas.doc.ic.ac.uk/downloads/manual.pdf>

[8] "NuSMV home page". 10 Jan 2016. <http://nusmv.fbk.eu/>

[9] "OpenMP.org." 2 June 2016. <http://openmp.org/wp/>

[10] "VAS – Verification of Autonomous Systems >> MCMAS". 18 Jan 2016. <http://vas.doc.ic.ac.uk/software/mcmas/>

[11] "Verics | VerICS". 18 Jan 2016. <http://verics.ipipan.waw.pl/>

[12] Ayewah, Nathaniel, et al. "Using FindBugs on production software." *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM, 2007.

[13] Baier, Christel, and Joost-Pieter Katoen. *Principles of model checking*. Vol. 26202649. Cambridge: MIT press, 2008.

[14] Ball, Thomas, Vladimir Levin, and Sriram K. Rajamani. "A decade of software model checking with SLAM." *Communications of the ACM* 54.7 (2011): 68-76.

[15] Basili, Victor R., and Richard W. Selby. "Comparing the effectiveness of software testing strategies." *Software Engineering, IEEE Transactions on* 12 (1987): 1278-1296.

[16] Bauland, Michael, et al. "The tractability of model-checking for LTL: The good, the bad, and the ugly fragments." *Electronic Notes in Theoretical Computer Science* 231 (2009): 277-292.

[17] Bernholtz, Orna, Moshe Y. Vardi, and Pierre Wolper. "An automata-theoretic approach to branching-time model checking." *Computer Aided Verification*. Springer Berlin Heidelberg, 1994.

[18] Biere, Armin, et al. "Bounded model checking." *Advances in Computers* 58 (2003): 117-148.

[19] Bjesse, Per. "What is formal verification?." *ACM SIGDA Newsletter* 35.24 (2005): 1.

[20] Bloem, Roderick, et al. "Symbolic implementation of alternating automata." *Implementation and Application of Automata*. Springer Berlin Heidelberg, 2006. 208-218.

[21] Boker, Udi, Orna Kupferman, and Adin Rosenberg. "Alternation removal in Büchi automata." *Automata, Languages and Programming*. Springer Berlin Heidelberg, 2010. 76-87.

[22] Bollig, Beate, and Lngo Wegener. "Improving the variable ordering of OBDDs is NP-complete." *Computers, IEEE Transactions on* 45.9 (1996): 993-1002.

[23] Bryant, Randal E. "Graph-based algorithms for boolean function manipulation." *Computers, IEEE Transactions on* 100.8 (1986): 677-691.

[24] Büchi, Julius Richard. "Weak Second-Order Arithmetic and Finite Automata." *Mathematical Logic Quarterly* 6.1-6 (1960): 66-92.

[25] Burmeister, Birgit, Afsaneh Haddadi, and Guido Matylis. "Application of multi-agent systems in traffic and transportation." *Software Engineering. IEE Proceedings.* Vol. 144. No. 1. IET, 1997.

[26] Cavada, Roberto, et al. "NuSMV 2.6 User Manual." 1 June 2016. <`http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`>

[27] Chaum, David. "The dining cryptographers problem: Unconditional sender and recipient untraceability." *Journal of Cryptology* 1.1 (1988): 65-75.

[28] Clarke, Edmund, Orna Grumberg, and Kiyoharu Hamaguchi. "Another look at LTL model checking." *Computer Aided Verification*. Springer Berlin Heidelberg, 1994.

[29] Clarke, Edmund M., et al. "Efficient generation of counterexamples and witnesses in symbolic model checking." *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*. ACM, 1995.

[30] Clarke, Edmund M., Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[31] Clarke, Edmund, et al. "Tree-like counterexamples in model checking." *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*. IEEE, 2002.

[32] Clarke, Edmund M., et al. "Model checking and the state explosion problem." *Tools for Practical Software Verification*. Springer Berlin Heidelberg, 2012. 1-30.

[33] Cormen, Thomas H., et al. *Introduction to Algorithms*. MIT Press, 2009.

[34] Darwiche, Adnan. "SDD: A new canonical representation of propositional knowledge bases." *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*. Vol. 22. No. 1. 2011.

[35] De Giacomo, Giuseppe, and Moshe Y. Vardi. "Linear temporal logic and linear dynamic logic on finite traces." *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence.* AAAI Press, 2013.

[36] De Giacomo, Giuseppe, et al. "Monitoring business metaconstraints based on LTL and LDL for finite traces." *Business Process Management.* Springer International Publishing, 2014. 1-17.

[37] Donaldson, Alastair. "Software Reliability – Compilers and Undefined Behaviour." Department of Computing, Imperial College London, 2015.

[38] Duret-Lutz, Alexandre, and Denis Poitrenaud. "SPOT: an extensible model checking library using transition-based generalized Büchi automata." *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on.* IEEE, 2004.

[39] Emerson, E. Allen, and Chin-Laung Lei. "Modalities for model checking (extended abstract): branching time strikes back." *Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages.* ACM, 1985.

[40] Emerson, E. Allen, and Joseph Y. Halpern. ""Sometimes" and "not never" revisited: on branching versus linear time temporal logic." *Journal of the ACM* (JACM) 33.1 (1986): 151-178.

[41] Faymonville, Peter, and Martin Zimmermann. "Parametric linear dynamic logic." *arXiv preprint* arXiv:1408.5957 (2014).

[42] Finch, Steven. "Central Binomial Coefficients." 29 April 2016. <`http://www.people.fas.harvard.edu/~sfinch/csolve/cbc.pdf`>

[43] Fritz, Carsten, and Thomas Wilke. "Simulation relations for alternating Büchi automata." Theoretical Computer Science 338.1-3 (2005): 275-314.

[44] Gartner. "Global Market Share Held by Semiconductor Vendors in 2015." *Statista – The Statistics Portal.* Statista. 10 Jan 2016. <`http://www.statista.com/statistics/266143/global-market-share-of-leading-semiconductor-vendors/`>

[45] Gastin, Paul, and Denis Oddoux. "Fast LTL to Büchi automata translation." *Computer Aided Verification.* Springer Berlin Heidelberg, 2001.

[46] Gopalan, Anandha. "Networks and Communications: Transport Layer." Department of Computing, Imperial College London, 2016. 28 May 2016. <`https://www.doc.ic.ac.uk/~axgopala/nac/slides/nac_03.pdf`>

[47] Harrison, John. "Formal Methods at Intel – An Overview." *Second NASA Formal Methods Symposium.* Vol. 8. 2010.

[48] Hodkinson, Ian. "140 Logic." Department of Computing, Imperial College London, 2015.

[49] Hojati, Ramin, Robert K. Brayton, and Robert P. Kurshan. "BDD-based debugging of designs using language containment and fair CTL." *Computer Aided Verification.* Springer Berlin Heidelberg, 1993.

[50] Holzmann, Gerard J. "The model checker SPIN." *IEEE Transactions on software engineering* 5 (1997): 279-295.

[51] Holzmann, Gerard J. "Landing a spacecraft on Mars." *Software*, IEEE 30.2 (2013): 83-86.

[52] Howell, Rodney R. "epsilon-NFA's and regular expressions." 24 April 2016. <`http://people.cis.ksu.edu/~rhowell/770s04/lectures/3-twoup.pdf`>

[53] Huang, Xiaowei, and Ron van der Meyden. "Symbolic Model Checking Algorithms for Temporal-Epistemic Logic."

[54] Huth, Michael, and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems.* Cambridge University Press, 2004.

[55] Jones, Andrew V., and Alessio Lomuscio. "A BDD-based BMC approach for the verification of multi-agent systems." *Proc. of CS&P.* Vol. 9. 2009.

[56] Kemerer, Chris F., and Mark C. Paulk. "The impact of design and code reviews on software quality: An empirical study based on PSP data." *Software Engineering, IEEE Transactions on* 35.4 (2009): 534-550.

[57] Kong, Jeremy, and Alessio Lomuscio. "MCMAS 1.2.2 – EG counterexample generation deadlock." Personal communication, 2016.

[58] Kouvaros, Panagiotis, and Alessio Lomuscio. "A counter abstraction technique for the verification of robot swarms." *Twenty-Ninth AAAI Conference on Artificial Intelligence.* 2015.

[59] Kwiatkowska, Marta, Gethin Norman, and David Parker. "PRISM 4.0: Verification of probabilistic real-time systems." *Computer Aided Verification.* Springer Berlin Heidelberg, 2011.

[60] Leucker, Martin, and César Sánchez. "Regular linear temporal logic." *Theoretical Aspects of Computing – ICTAC 2007.* Springer Berlin Heidelberg, 2007. 291-305.

[61] Leveson, Nancy G., and Clark S. Turner. "An investigation of the Therac-25 accidents." *Computer* 26.7 (1993): 18-41.

[62] Lomuscio, Alessio, and Mark Ryan. "On the relation between interpreted systems and Kripke models." *Agents and Multi-Agent Systems Formalisms, Methodologies, and Applications* (1998): 46-59.

[63] Lomuscio, Alessio, and Marek Sergot. "Deontic interpreted systems." *Studia Logica* 75.1 (2003): 63-92.

[64] Lomuscio, Alessio, and Franco Raimondi. "Model checking knowledge, strategies, and games in multi-agent systems." *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems.* ACM, 2006.

[65] Lomuscio, Alessio, Hongyang Qu, and Franco Raimondi. "MCMAS: an open-source model checker for the verification of multi-agent systems." *International Journal on Software Tools for Technology Transfer* (2015): 1-22.

[66] Lomuscio, Alessio. "C303: Systems verification." Department of Computing, Imperial College London, 2015.

[67] Lomuscio, Alessio, and Hugo Paquet. "Verification of Multi-Agent Systems via SDD-based Model Checking." *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[68] Maidi, Madjid. "The common fragment of CTL and LTL." *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on.* IEEE, 2000.

[69] McMillan, Kenneth Lauchlin. "Symbolic model checking: an approach to the state explosion problem." (1992).

[70] Meski, Artur, Wojciech Penczek, and Maciej Szreter. "BDD-based Bounded Model Checking for LTLK over Two Variants of Interpreted Systems." *Proc. of LAM* (2012): 35-50.

[71] Meski, Artur, et al. "BDD-versus SAT-based bounded model checking for the existential fragment of linear temporal logic with knowledge: algorithms and their performance." *Autonomous Agents and Multi-Agent Systems* 28.4 (2014): 558-604.

[72] Milvang-Jensen, Kim, and Alan J. Hu. "BDDNOW: a parallel BDD package." *Formal Methods in Computer-Aided Design*. Springer Berlin Heidelberg, 1998.

[73] Miyano, Satoru, and Takeshi Hayashi. "Alternating finite automata on ω-words." *Theoretical Computer Science* 32.3 (1984): 321-330.

[74] Nicely, Thomas R. "Pentium FDIV Flaw FAQ." *Some Results of Research in Computational Number Theory* (2008).

[75] Porter, Timothy. "Interpreted systems and Kripke models for multiagent systems from a categorical perspective." *Theoretical computer science* 323.1 (2004): 235-266.

[76] Raimondi, Franco, and Alessio Lomuscio. "A tool for specification and verification of epistemic properties in interpreted systems." *Electronic Notes in Theoretical Computer Science* 85.2 (2004): 176-191.

[77] Rogers, Alex, et al. "The effects of proxy bidding and minimum bid increments within eBay auctions." *ACM Transactions on the Web (TWEB)* 1.2 (2007): 9.

[78] Rozier, Kristin Y., and Moshe Y. Vardi. "LTL satisfiability checking." *Model checking software*. Springer Berlin Heidelberg, 2007. 149-167.

[79] Rozier, Kristin Y. "Linear temporal logic symbolic model checking." *Computer Science Review* 5.2 (2011): 163-203.

[80] Rozier, Kristin Y., and Moshe Y. Vardi. "A Multi-Encoding Approach for LTL Symbolic Satisfiability Checking." 14 May 2016. <`http://www.cs.rice.edu/~vardi/papers/fm11b.pdf`>

[81] Rudell, Richard. "Dynamic variable ordering for ordered binary decision diagrams." *Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society Press, 1993.

[82] Russell, Stuart, and Peter Norvig. "Artificial intelligence: a modern approach." (1995).

[83] Savnik, Iztok. "Index data structure for fast subset and superset queries." *Availability, Reliability, and Security in Information Systems and HCI*. Springer Berlin Heidelberg, 2013. 134-148.

[84] Schnoebelen, Philippe. "The Complexity of Temporal Logic Model Checking." *Advances in modal logic* 4.393-436 (2002): 35.

[85] Schuppan, Viktor, and Armin Biere. "Shortest Counterexamples for Symbolic Model Checking of LTL with Past." *Tools and Algorithms for the Construction and Analysis of Systems* (2005): 493-509.

[86] Somenzi, Fabio. "Binary Decision Diagrams." *Calculational System Design* 173 (1999): 303.

[87] Somenzi, Fabio, and Roderick Bloem. "Efficient Büchi automata from LTL formulae." *Computer Aided Verification*. Springer Berlin Heidelberg, 2000.

[88] Somenzi, Fabio. "CUDD: CU decision diagram package-release 2.4. 0." University of Colorado at Boulder, 2009.

[89] Thompson, Ken. "Programming techniques: Regular expression search algorithm." *Communications of the ACM* 11.6 (1968): 419-422.

[90] van Bakel, Steffen. "Discrete Mathematics Lecture Notes: Part I." Department of Computing, Imperial College London, 2015.

[91] van Dijk, Tom, Alfons Laarman, and Jaco van de Pol. "Multi-core BDD operations for symbolic reachability." *Electronic Notes in Theoretical Computer Science* 296 (2013): 127-143.

[92] van Ditmarsch, Hans, Jan van Eijck, and William Wu. "One Hundred Prisoners and a Lightbulb – Logic and Computation." *Twelfth International Conference on the Principles of Knowledge Representation and Reasoning*. 2010.

[93] Vardi, Moshe. "Alternating automata and program verification." *Computer Science Today* (1995): 471-485.

[94] Vardi, Moshe Y."An automata-theoretic approach to linear temporal logic." *Logics for concurrency*. Springer Berlin Heidelberg, 1996. 238-266.

[95] Vardi, Moshe Y. "Branching vs. linear time: Final showdown." *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2001. 1-22.

[96] Vardi, Moshe Y. "The rise and fall of LTL." *GandALF*. EPTCS 54 (2011).

[97] Vega, Frank. "NP versus PSPACE." 24 May 2016. <`https://hal.archives-ouvertes.fr/hal-01196489/file/NP-versus-PSPACE.pdf`>

[98] Weinert, Alexander, and Martin Zimmermann. "Visibly Linear Dynamic Logic." *arXiv preprint arXiv:1512.05177* (2015). 30 May 2016. <`http://arxiv.org/pdf/1512.05177v1.pdf`>

[99] Wolper, Pierre. "Temporal logic can be more expressive." *Information and Control* 56.1 (1983): 72-99.

[100] Wozna, Bozena, and Andrzej Zbrzezny. "VerICS 2006-a Model Checker for Real-Time and Multi-Agent Systems."

Appendix A

ISPL Model for the Bit Transmission Protocol

This example is taken from [7], with minor adaptations (specifically, removing the fairness constraint, allowing the channel to start in any state, and adding several `Evaluation` variables).

```
1  -- The Bit transmission problem in ISPL ( from Fagin et al., Reasoning
2  -- about knowledge, 1995).
3  -- Basic case, no faulty behaviour for Receiver.
4
5  Agent Environment
6    Vars:
7      state : {S,R,SR,none};
8    end Vars
9    Actions = {S,SR,R,none};
10   Protocol:
11     state=S: {S,SR,R,none};
12     state=R: {S,SR,R,none};
13     state=SR: {S,SR,R,none};
14     state=none: {S,SR,R,none};
15   end Protocol
16   Evolution:
17     state=S if (Action=S);
18     state=R if (Action=R);
19     state=SR if (Action=SR);
20     state=none if (Action=none);
21   end Evolution
22 end Agent
23
24 Agent Sender
25   Vars:
26     bit : { b0, b1}; -- The bit can be either zero or one
27     ack : boolean;  -- This is true when the ack has been received
28   end Vars
29   Actions = { sb0,sb1,nothing };
30   Protocol:
31     bit=b0 and ack=false : {sb0};
32     bit=b1 and ack=false : {sb1};
33     ack=true : {nothing};
34   end Protocol
35   Evolution:
36     (ack=true) if (ack=false) and
37        ( ( (Receiver.Action=sendack) and (Environment.Action=SR) )
38         or
39        ( (Receiver.Action=sendack) and (Environment.Action=R) )
```

```
40        );
41    end Evolution
42  end Agent
43
44  Agent Receiver
45    Vars:
46      state : { empty, r0, r1 };
47    end Vars
48    Actions = {nothing,sendack};
49    Protocol:
50      state=empty : {nothing};
51      (state=r0 or state=r1): {sendack};
52    end Protocol
53    Evolution:
54      state=r0 if ( ( (Sender.Action=sb0) and (state=empty) and
55                      (Environment.Action=SR) ) or
56                    ( (Sender.Action=sb0) and (state=empty) and
57                      (Environment.Action=S) ) );
58      state=r1 if ( ( (Sender.Action=sb1) and (state=empty) and
59                      (Environment.Action=SR) ) or
60                    ( (Sender.Action=sb1) and (state=empty) and
61                      (Environment.Action=S) ) );
62    end Evolution
63  end Agent
64
65  Evaluation
66    recbit if ( (Receiver.state=r0) or (Receiver.state=r1) );
67    recack if ( ( Sender.ack = true ) );
68    bit0 if ( (Sender.bit=b0));
69    bit1 if ( (Sender.bit=b1) );
70    envworks if ( Environment.state=SR );
71    tt if ( (Sender.bit = b0) or (Sender.bit = b1));
72  end Evaluation
73
74  InitStates
75    ( (Sender.bit=b0) or (Sender.bit=b1) ) and
76    ( Receiver.state=empty ) and ( Sender.ack=false);
77  end InitStates
78
79  Groups
80    g1 = {Sender,Receiver};
81  end Groups
82
83  Formulae
84    -- The bit is always eventually received.
85    AF recbit;
86    LTL F recbit;
87    CTL* A(F recbit);
88    LDL <tt*> recbit;
89    CDL* A(<tt*> recbit);
90
91    -- If the sender receives an ack, he knows the receiver knows
92    -- the value of the bit.
93    AG (recack -> K(Sender, K(Receiver, bit0) or K(Receiver, bit1)));
94    LTL G(recack -> K(Sender, K(Receiver, bit0) or K(Receiver, bit1)));
95    CTL* A(G(recack -> K(Sender, K(Receiver, bit0) or K(Receiver, bit1))));
96    LDL [tt*](recack -> K(Sender, K(Receiver, bit0) or K(Receiver, bit1)));
97    CDL* A([tt*](recack -> K(Sender, K(Receiver, bit0) or K(Receiver, bit1))));
98
99    -- It's possible for the bit to eventually be received.
```

```
100   EF recbit;
101   CTL* E(F recbit);
102   CDL* E(<tt*> recbit);
103
104   -- Always possible to eventually receive the bit.
105   AG EF recbit;
106   CTL* A(G(E(F(recbit))));
107   CDL* A([tt*](E(<tt*>(recbit))));
108
109   -- On every path, if the bit is received then an ack is received.
110   LTL (F recbit) -> (F recack);
111   LDL (<tt*> recbit) -> (<tt*> recack);
112   CDL* A((<tt*> recbit) -> (<tt*> recack));
113
114   -- If the environment works on every other step, then the ack is
115   -- eventually received.
116   LDL ([(tt;tt)*] envworks) -> (<tt*> recack);
117   CDL* A (([(tt;tt)*] envworks) -> (<tt*> recack));
118
119   -- If the environment works on every other step, then the ack is
120   -- eventually received on an odd / even step. Both false
121   -- Note that we don't say anything about the environment's behaviour
122   -- on odd steps.
123   LDL ([(tt;tt)*] envworks) -> (<(tt;tt)*> (!recack and <tt>recack));
124   LDL ([(tt;tt)*] envworks) -> (<tt;(tt;tt)*> (!recack and <tt>recack));
125 end Formulae
```

Appendix B

ISPL Model for the Go-Back-N Protocol

Please see Section 8.8 for a greater discussion of the details of this protocol. This instance has a buffer size $M = 3$, and a window size $N = 2$.

```
1  -- The Go-Back-N ARQ protocol.
2  -- This instance has a buffer size of 3 and window size of 2
3  -- Generated by Jeremy Kong
4  Semantics = SingleAssignment;
5
6  Agent Environment
7    Vars:
8      state: {S, R, SR, none};
9      s_r_1: {b0, b1, empty};
10     s_r_1_seq: 0..3;
11     r_s_1: 0..3;
12     s_r_2: {b0, b1, empty};
13     s_r_2_seq: 0..3;
14     r_s_2: 0..3;
15     s_r_3: {b0, b1, empty};
16     s_r_3_seq: 0..3;
17     r_s_3: 0..3;
18   end Vars
19
20   Actions = {S, R, SR, none};
21   Protocol:
22     Other: {S, R, SR, none};
23   end Protocol
24
25   Evolution:
26     state=S if Action=S;
27     state=SR if Action=SR;
28     state=R if Action=R;
29     state=none if Action=none;
30     s_r_3=s_r_2 if (Action=SR or Action=S);
31     s_r_3=empty if (Action=R or Action=none);
32     s_r_2=s_r_1 if (Action=SR or Action=S);
33     s_r_2=empty if (Action=R or Action=none);
34     s_r_1=empty if (Sender.Action=none);
35     s_r_1=b0 if (
36       (Sender.Action=b10) or
37       (Sender.Action=b20) or
38       (Sender.Action=b30)
39     );
```

```
40      s_r_1=b1 if (
41        (Sender.Action=b11) or
42        (Sender.Action=b21) or
43        (Sender.Action=b31)
44      );
45      s_r_1_seq=1 if (Sender.Action=b10 or Sender.Action=b11);
46      s_r_1_seq=2 if (Sender.Action=b20 or Sender.Action=b21);
47      s_r_1_seq=3 if (Sender.Action=b30 or Sender.Action=b31);
48      s_r_1_seq=3 if (Sender.Action=none);
49      s_r_2_seq=s_r_1_seq if (Action=none) or !(Action=none);
50      s_r_3_seq=s_r_2_seq if (Action=none) or !(Action=none);
51      r_s_3=r_s_2 if (Action=S or Action=SR);
52      r_s_3=0 if (Action=R or Action=none);
53      r_s_2=r_s_1 if (Action=S or Action=SR);
54      r_s_2=0 if (Action=R or Action=none);
55      r_s_1=0 if (Receiver.Action=null);
56      r_s_1=1 if (Receiver.Action=r1);
57      r_s_1=2 if (Receiver.Action=r2);
58      r_s_1=3 if (Receiver.Action=r3);
59    end Evolution
60
61  end Agent
62
63  Agent Sender
64    Lobsvars={s_r_1, r_s_3, s_r_1_seq};
65    Vars:
66      done: boolean;
67      recack: 0..3;
68      dispatch: 0..3;
69      value1: {b0, b1};
70      value2: {b0, b1};
71      value3: {b0, b1};
72    end Vars
73    Actions = {none, b10, b11, b20, b21, b30, b31};
74    Protocol:
75      done=true : {none};
76      dispatch=1 and value1=b0 : {b10};
77      dispatch=1 and value1=b1 : {b11};
78      dispatch=2 and value2=b0 : {b20};
79      dispatch=2 and value2=b1 : {b21};
80      dispatch=3 and value3=b0 : {b30};
81      dispatch=3 and value3=b1 : {b31};
82      Other: {none};
83    end Protocol
84    Evolution:
85      done=true if (Environment.r_s_3=3);
86      (recack=1) if (Environment.r_s_3=1);
87      (recack=2) if (Environment.r_s_3=2);
88      (recack=3) if (Environment.r_s_3=3);
89      (dispatch=dispatch+1) if (done=false) and
90                               (recack + 2 > dispatch) and
91                               (dispatch < 3);
92      (dispatch=recack+1) if (done=false) and
93                             (Environment.r_s_3=recack) or
94                             !((recack + 2 > dispatch) and (dispatch < 3));
95      (dispatch=dispatch) if (done=true);
96    end Evolution
97  end Agent
98
99  Agent Receiver
```

```
  Lobsvars={r_s_1, s_r_3, s_r_3_seq};
  Vars:
    recv1 : {empty, r0, r1};
    recv2 : {empty, r0, r1};
    recv3 : {empty, r0, r1};
  end Vars
  Actions = {null, r1, r2, r3};
  Protocol:
    recv1=empty or Environment.s_r_3=empty : {null};
    recv1<>empty and recv2=empty and Environment.s_r_3<>empty : {r1};
    recv2<>empty and recv3=empty and Environment.s_r_3<>empty : {r2};
    recv3<>empty and Environment.s_r_3<>empty: {r3};
  end Protocol
  Evolution:
    recv1=r0 if (Environment.s_r_3=b0) and (recv1=empty) and
                (Environment.s_r_3_seq=1);
    recv1=r1 if (Environment.s_r_3=b1) and (recv1=empty) and
                (Environment.s_r_3_seq=1);
    recv2=r0 if (Environment.s_r_3=b0) and (recv2=empty) and
                (Environment.s_r_3_seq=2);
    recv2=r1 if (Environment.s_r_3=b1) and (recv2=empty) and
                (Environment.s_r_3_seq=2);
    recv3=r0 if (Environment.s_r_3=b0) and (recv3=empty) and
                (Environment.s_r_3_seq=3);
    recv3=r1 if (Environment.s_r_3=b1) and (recv3=empty) and
                (Environment.s_r_3_seq=3);
  end Evolution
end Agent

Evaluation
  tt if ( Environment.state = SR ) or ! ( Environment.state = SR );
  mismatch1 if (Receiver.recv1=r0 and Sender.value1=b1) or
               (Receiver.recv1=r1 and Sender.value1=b0);
  mismatch2 if (Receiver.recv2=r0 and Sender.value2=b1) or
               (Receiver.recv2=r1 and Sender.value2=b0);
  mismatch3 if (Receiver.recv3=r0 and Sender.value3=b1) or
               (Receiver.recv3=r1 and Sender.value3=b0);
  rbit30 if (Receiver.recv3=r0);
  rbit31 if (Receiver.recv3=r1);
  envworks if (Environment.state=SR);
  envbroken if (Environment.state=none);
end Evaluation

InitStates
  (
    Receiver.recv1=empty and
    Receiver.recv2=empty and
    Receiver.recv3=empty and
    Sender.done = false and
    Sender.recack = 0 and
    Sender.dispatch = 1 and
    Environment.s_r_1=empty and
    Environment.s_r_2=empty and
    Environment.s_r_3=empty and
    Environment.r_s_1=0 and
    Environment.r_s_2=0 and
    Environment.r_s_3=0
  );
end InitStates
```

```
160 Groups
161   g1 = {Sender, Receiver};
162 end Groups
163
164 Formulae
165   LDL [tt*](
166     !mismatch1 and
167     !mismatch2 and
168     !mismatch3
169   );
170   LDL ([tt*] envworks) ->
171       (<tt*> GK(g1, K(Receiver, rbit30) or K(Receiver, rbit31)));
172   CDL* envworks -> E(<((envworks; envbroken) + (envworks; envworks; envbroken))*>
173                       GK(g1, K(Receiver, rbit30) or K(Receiver, rbit31)));
174 end Formulae
```

Appendix C

Additional Proofs

Lemma C.1. Suppose that for $i \in 0, \ldots, k$ we have $f(y_i, n) = O(n2^{y_i})$ and $0 \leq y_i < x$. Further suppose that $f(x, n) = \left(\sum_{i=0}^{k} f(y_i, n)\right) + O(n2^x)$. Then, $f(x, n) = O(n2^x)$.

Proof. From the definition of big-O notation we have for each i, $f(y_i, n) \leq c_i n 2^{y_i}$ for some positive c_i. Then, observe that

$$
\begin{aligned}
f(x,n) &= \left(\sum_{i=0}^{k} f(y_i, n)\right) + O(n2^x) \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{y_i}\right) + O(n2^x) && \text{by above assertion} \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{y_i}\right) + cn2^x && \text{for some positive } c\text{, by definition} \\
&\leq \left(\sum_{i=0}^{k} c_i n 2^{x}\right) + cn2^x && \text{since for each } i,\ 0 \leq y_i < x \text{ and so } 2^{y_i} \leq 2^x \\
&= \left(\left(\sum_{i=0}^{k} c_i\right) + c\right) n2^x
\end{aligned}
$$

Furthermore, since every c_i and c are positive, their sum is positive. Thus $f(x, n) = O(n2^x)$.