Department of Computing
Imperial College London

MEng Individual Project

A Model Checker for Strategy Logic

Author:
Petr Čermák

Supervisor:
Prof. Alessio R. Lomuscio

June 2014

Abstract

There is a gap between game theory and practical model checking. Although logics for expressing game-theoretic concepts like Nash equilibria, notably Strategy Logic (SL), have been put forward recently, there are currently no tools supporting them. This report documents the design of the first practical model checking algorithms for two fragments of SL and their symbolic implementation using binary decision diagrams. We explain how we incorporated both algorithms in the existing model checker MCMAS and added support for strategy synthesis. We evaluate the performance of the algorithms on several scalable scenarios and compare it with the original tool. Our results demonstrate that SL model checking is feasible in practice despite its high computational complexity.

Acknowledgements

I would like to thank:

- Professor Alessio Lomuscio for his invaluable guidance throughout the whole project,
- Professor Aniello Murano and Dr. Fabio Mogavero for their feedback on the algorithms and their help with finding scalable scenarios for evaluation,
- the Department of Computing for funding my trips to the *2nd International Workshop on Strategic Reasoning* and the *26th International Conference on Computer Aided Verification*,
- my family and friends, who have always been there for me, and
- my girlfriend Klára for her love and support.

Contents

Chapter 1

Introduction

Computer systems have an ever-increasing presence in our everyday lives. We rely on them in almost all activities both at work and during our free time. It will probably not be long until we meet driverless cars on the road and receive deliveries from unmanned quadcopters [19,24]. It seems that more and more decisions in the future will be made by computers without any human intervention. Yet computers are infamous for their frequent faults. Each of us has witnessed freezing applications countless times. While the usual type of damage caused by a computer bug today is the loss of several hours of unsaved work, autonomous system failures in the future might result in thousands of deaths in the blink of an eye. The six accidents caused by massive radiation overdoses by the Therac-25 medical electron accelerator between 1985 and 1987 remind us that such scenarios are not science fiction [59]. It should be clear that safety-critical systems need to be verified before being deployed in order to avoid loss of life and money. Fortunately, formal verification is now finally becoming the norm in many areas including processor design [53].

Model-checking is a popular approach to systems verification [62]. It entails translating the system and the property to be verified to a model $\mathcal{M}$ and a formula φ respectively and then checking that the model satisfies the formula, i.e. that $\mathcal{M} \models \varphi$ holds. This approach can be applied in many areas including reactive systems, which never terminate, by modelling them as multi-agent systems, in which agents with local states interact with each other via synchronous actions. Various frameworks for modelling such systems are discussed in Section 2.1.

Several temporal logics for specifying system properties have been proposed, including Linear Temporal Logic (Ltl) [80], Computation Tree Logic (Ctl) [33], and Full Branching Time Logic (Ctl*) [37]. More recently, Alternating-Time Logic (Atl) and Full Alternating-Time Logic (Atl*) [18] were introduced. Atl and Atl* are often more suitable for multi-agent systems since their operators refer to properties achievable by given groups of agents. Intuitively, an Atl formula $\langle\!\langle\{\alpha, \beta\}\rangle\!\rangle\varphi$ expresses that *"agents α and β can together enforce φ"*. A detailed discussion of these logics can be found in Section 2.2. All of them except Atl*, which has a high model checking complexity, are now supported by multiple toolkits (see Section 2.5).

However, neither Atl nor Atl* syntax refers to strategies directly. Hence, the logics cannot express properties such as *"agents α and β can enforce φ while sharing the same strategy"*. A new formalism, *Strategy Logic* (Sl), which treats strategies explicitly via first-order quantifiers $\langle\!\langle x\rangle\!\rangle$, $[\![x]\!]$ and agent binding (a, x), has thus been introduced in [76]. Sl strictly subsumes all logics in the Atl* hierarchy and can express game-theoretic concepts like Nash Equilibria. The aforementioned property, inexpressible in Atl*, can be written in Sl as $\langle\!\langle x\rangle\!\rangle(\alpha, x)(\beta, x)\varphi$. However, the price for the large expressiveness of Sl includes NonElementarySpace-hard model checking complexity [72], undecidability of satisfiability [77], and non-behavioural strategies [75]. We are not aware of any existing model checker that would support Sl.

In order to avoid the problems associated with full Sl, several syntactic fragments have been proposed [72]. The least expressive subset, *One-Goal Strategy Logic* (Sl[1g]), has the same model checking complexity as Atl*, namely 2ExpTime-complete with respect to the size of the formula, while still strictly subsuming it. Moreover, Sl[1g] strategies are behavioural and can therefore be synthesised [75]. We have designed a novel model checking algorithm for Sl[1g] by reducing the problem to solving a two-player parity game. Our algorithm, presented in Chapter 5, admits an efficient symbolic implementation and supports general strategy synthesis. Moreover, we prove that it is correct and has optimal worst-case time complexity.

We introduce and investigate another variant of SL, *Epistemic Strategy Logic*, or Strategy Logic with Knowledge (SLK). Unlike SL and SL[1G], we define SLK on strategies with imperfect recall (i.e. memoryless strategies where an agent's action is determined only by its current local state) and supports epistemic modalities expressing agents' knowledge. We show that the theoretical complexity of SLK model checking is PSPACE with respect to both the size of the model and the formula. We then provide an efficient model checking algorithm for SLK which can be implemented symbolically and supports witness/counterexample strategy synthesis. We also prove that it is correct. Both the logic and the algorithm are discussed in Chapter 4.

MCMAS is a BDD-based model checker for the verification of multi-agent systems developed at Imperial College London [63]. The specification languages it currently supports include CTL and ATL with epistemic and deontic modalities. MCMAS and other existing model checking tools are discussed in Section 2.5. Various verification methods including BDDs are explained in Section 2.3.

We implemented the novel model checking algorithms for both SLK and SL[1G] as extensions for MC-MAS and thereby extended the set of logics it supports. Both extensions support witness/counterexample strategy synthesis and the SL[1G] extension also supports general strategy synthesis. An overview of the existing MCMAS architecture as well as the usage and implementation of our algorithms is provided in Chapter 6. The experimental results we obtained using our extensions are presented in Section 6.4.

1.1 Objectives

Our goal was to *implement the first model checker for* SL. We believed that it would be a very important milestone for the development of SL. It would demonstrate the potential of the logic, provide a solid baseline for future research, and promote the synergy of systems verification and game theory. In order to achieve this, our objectives were to:

1. **Research SL and its fragments.** SL is a relatively new formalism for specifying properties in systems verification introduced in 2010 [76]. The first aim of this project was therefore to research SL as well as its syntactic fragments in order to have a good understanding of its expressiveness, complexity, decidability, and other properties. In other words, it was necessary to obtain basic knowledge of the research area to be able to set challenging yet achievable objectives for the project.

2. **Design an efficient model checking algorithm.** The main aim of this project was to come up with a model checking algorithm for SL or some of its variants. While theoretical automata-based model checking procedures have been proposed [72], it is important to design novel algorithms which use efficient data structures (e.g. binary decision diagrams discussed in Subsection 2.3.1) in order to reduce the effects of the high computational complexity of the problem.

3. **Implement the algorithm as a tool.** The next objective was to create a tool implementing the model checking algorithm we designed. The implementation of the algorithm had to be robust, efficient, and user-friendly. Moreover, the source code had to have a good design and appropriate documentation as we intended to release it as open source.

   A standalone tool would have to support the whole verification framework including a modelling language parser, state space encoding, reachability calculation, etc. In order to avoid "reinventing the wheel", we could extend an existing tool instead (several existing tools are presented in Section 2.5). This would give us more time to add more functionality and/or optimise the performance of the algorithm.

4. **Support strategy synthesis.** In addition to model checking, i.e. determining whether a given specification is true or false, we also wanted to synthesise the corresponding strategies for the agents. These strategies would provide us with a deeper insight into why a particular formula holds or does not hold. More importantly, strategy synthesis would make our tool much more useful as it would allow users to underspecify the agents' behaviour and have it automatically generated.

1.2 Challenges

Overall, the project was very challenging due to its novelty. The work entailed transforming high-level theoretical concepts into low-level programming code with almost no prior experience in the field. The biggest challenges encountered throughout the project were:

1. **No existing algorithms.** While there has been a lot of focus on the theoretical aspects of SL in the past few years [72, 73, 75, 76], no practical model checking algorithm has been proposed or implemented for either SL or any of its fragments so far. In fact, we are not aware of any existing tool that would support ATL*, which was introduced in 2002 [18] and is strictly subsumed even by the least expressive fragment of SL. Therefore, it was very difficult to find any starting points for our research into model checking algorithms.

2. **Undecidability.** SL is a very powerful formalism for expressing complex properties of multi-agent systems. While this is desirable from the user's perspective, it makes designing and implementing efficient algorithms for handling the logic very difficult. Moreover, certain problems are impossible to solve due to undecidability. Unfortunately, this was the case when we considered adding incomplete information to the original SL, as described in Chapter 3. In order to achieve decidability, we had to consider less expressive variants of SL instead.

3. **Complexity.** The high model checking complexity of SL has a severe impact on performance. Firstly, the number of temporal operators within a formula we can handle is limited since both SLK and SL[1G] have high complexity with respect to the size of the formula, namely PSPACE and 2EXPTIME-COMPLETE. However, it turned out that that the PSPACE complexity of SLK with respect to the size of the model is an even bigger problem, rendering SLK model checking feasible only for very small state spaces in certain scenarios (see Section 6.4).

4. **Existing codebase.** MCMAS is one of the leading model checkers (see Subsection 2.5.2). However, despite being developed as open source, it is very badly designed from a software engineering point of view. The problems include a complete lack of testing, violation of OOP principles, frequent usage of global variables, and inconsistent code style. The tightly coupled code design made it very difficult for us to test our new functionality. Unfortunately, the lack of testing makes refactoring existing code very frustrating. In our opinion, the tool has outlived its original purpose and should be rewritten from scratch.

1.3 Contributions

We have designed and implemented model checking algorithms for two fragments of SL. We believe this is an important contribution to the research area, as our tool is the first model checker to support SL or any of its subsets. Throughout this project, we have developed:

1. **New SL fragment.** We introduced a new fragment of SL called *Epistemic Strategy Logic* or Strategy Logic with Knowledge (SLK). Unlike SL, SLK is defined with respect to imperfect recall semantics (i.e. agents have no memory of the past) and supports epistemic modalities expressing agents' knowledge (see Subsection 2.2.6). Hence, SLK specifications can represent complex game-theoretic concepts like Nash equilibria under incomplete information.

   We give the syntax and semantics of SLK. We also discuss its limitations and prove that the SLK model checking problem belongs to the PSPACE complexity class with respect to both the size of the model and the formula (see Subsection 4.2.1).

   Section 4.1 provides a detailed description of SLK.

2. **Model checking algorithm for SLK.** We have designed a novel model checking algorithm for SLK which has worst-case exponential time complexity both in the size of the model and the formula. We prove its correctness and provide an efficient symbolic encoding of it using binary decision diagrams (see Subsections 2.3.1 and 2.3.2). In addition, we explain how the algorithm (and its symbolic encoding) can be used for the purposes of witness/counterexample strategy synthesis.

   Section 4.2 provides a detailed description of the algorithm and its symbolic representation.

3. **Model checking algorithm for** Sl[1g]**.** We describe a novel algorithm which reduces the model checking problem for Sl[1g] to the problem of solving two-player parity games. Moreover, we prove its correctness, show that it is optimal, and explain how it can be used to synthesise strategies for arbitrary Sl[1g] formulas. Again, we also provide an efficient symbolic encoding of the algorithm.

   We believe that this is the *biggest achievement of the project* as Sl[1g] subsumes Atl*, which (as already mentioned in Section 1.2) was introduced in 2002 [18] but, as far as we know, has no practical implementation yet. Since Atl* has the same model checking complexity as Sl[1g], our algorithm is also optimal for it.

   Chapter 5 provides a detailed description of Sl[1g], the algorithm, and its symbolic representation.

4. **Implementation of both algorithms.** We have extended the functionality of the MCMAS model checker (discussed in Subsection 2.5.2) with support for both Slk and Sl[1g]. We also implemented strategy synthesis for both fragments. The tool will be released as open source[1] so that it could be further developed in the future.

   Chapter 6 provides a detailed description of the existing MCMAS architecture and our extensions.

5. **Experimental evaluation.** We evaluated the performance of our Slk and Sl[1g] MCMAS extensions on several scalable scenarios. Moreover, we demonstrated the expressiveness of the fragments as well as the newly implemented functionality by automatically synthesising strategies that enforce properties which were not supported by the original tool (e.g. we generate a strategy for a scheduler which ensures the existence of Nash equilibria, see Subsection 6.4.2).

   The scenarios and the experimental results of our implementation are discussed in Section 6.4.

A detailed evaluation of the project can be found in Chapter 7.

1.4 Published Work

While working on the project, we published a tool paper on Slk implementation in MCMAS [27]. The paper was a joint work with my supervisor, Professor Alessio Lomuscio, and Dr. Fabio Mogavero and Professor Aniello Murano from the Computer Science Division (Sezione di Informatica) of the Department of Physical Science (Dipartimento di Scienze Fisiche) of the University of Naples "Federico II" (Università degli Studi di Napoli "Federico II"), Italy. The paper was accepted at the *26th International Conference on Computer Aided Verification 2014*. It summarises the procedure described in Section 4.2, its implementation provided in Section 6.2, and the experimental results presented in Section 6.4.

Given its novelty, we intend to publish a paper on our model checking algorithm for Sl[1g] in the summer (again as a joint work).

[1]The Slk extension, called MCMAS-SLK, has already been released under GNU General Public License (GPL) [5] and is available at `http://vas.doc.ic.ac.uk/software/tools/`.

Chapter 2

Background

In this chapter we describe concepts and methods from the area of software verification which the rest of this report is based on. We also present a list of existing tools related to our work. This chapter can be skipped at first reading and used as a reference in the subsequent chapters.

The ultimate goal of our project is to develop a tool that can verify SL specifications on multi-agent systems. Formal verification techniques typically comprise three components [49]:

1. a *framework for modelling systems*, typically a description language of some sort;
2. a *specification language* for describing the properties to be verified;
3. a *verification method* to establish whether the description of a system satisfies the specification.

Let us consider a system S and a property P to be verified (on the system). There are two main approaches to verification [49]:

- In *proof-based verification*, the system description is a set of formulas Γ_S and the property to be verified is another formula φ_P. The verification method consists of trying to prove $\Gamma_S \vdash \varphi_P$. This usually requires guidance and expertise from the user.
- In *model-based verification*, the system description is a model $\mathcal{M}_S$ and the property to be verified is a formula ϕ_P. The verification method consists of computing whether $\mathcal{M}_S$ satisfies ϕ_P ($\mathcal{M}_S \models \phi_P$). This can usually be done automatically for finite models.

In this project we focus purely on a model-based verification method called *model checking*, which refers to the process of determining whether a formula ϕ_P holds in a concrete model $\mathcal{M}_S$ [49]:

$$\mathcal{M}_S \overset{?}{\models} \phi_P$$

A *model checker* is a tool (i.e. a computer program) which performs this calculation automatically.

We will now describe various approaches to each of the three components of formal verification using model checking: how the system S is represented as a model $\mathcal{M}_S$, how the property P is expressed as a formula ϕ_P, and how the answer ($\mathcal{M}_S \models \phi_P$, or $\mathcal{M}_S \not\models \phi_P$) is determined.

2.1 Frameworks for Modelling Systems

Given a system S, we aim to represent it using a model $\mathcal{M}_S$. We discuss here three formalisms of increasing complexity: Kripke models, concurrent game structures, and interpreted systems.

2.1.1 Kripke Models

Kripke semantics are arguably the most popular modal semantics nowadays [48, 55]. The basic idea of Kripke semantics is that a formula is true at some worlds, and false at others [62]. The semantics are based on two concepts: Kripke frames and Kripke models. Intuitively, a *Kripke frame* represents a set of worlds together with transitions between them.

Definition 2.1 (Kripke Frames)**.** Let W be a non-empty set of worlds and $R \subseteq W \times W$ a binary *accessibility relation* on it. Then $\mathcal{F} \triangleq (W, R)$ is a *Kripke frame.*

For example, if we wanted to model a pedestrian traffic light as a Kripke frame $\mathcal{F}_{\text{TL}} = (W_{\text{TL}}, R_{\text{TL}})$, it would have two possible states $W_{\text{TL}} = \{\text{red}, \text{green}\}$. Assuming that the light changes at every temporal step, the accessibility relation of $\mathcal{F}_{\text{TL}}$ would be $R_{\text{TL}} = \{(\text{red}, \text{green}), (\text{green}, \text{red})\}$. In order to be able to specify properties of the system, we need to fix a set of atomic propositions and specify the set of worlds where each of them holds. This is done by adding an assignment function to the Kripke frame. The result is a *Kripke model*:

Definition 2.2 (Kripke Models)**.** Let $\mathcal{F} = (W, R)$ be a Kripke frame, AP a finite non-empty set of atomic propositions, and $h : AP \to 2^W$ an *assignment* into $\mathcal{F}$. Then $\mathcal{M} \triangleq (\mathcal{F}, h)$ is a *Kripke model.*

Suppose we wanted to express a property regarding the current colour of the traffic light. We would introduce two atoms $AP \triangleq \{\mathit{green}, \mathit{red}\}$ and use an assignment h_{TL} defined as $h_{\text{TL}}(\mathit{green}) \triangleq \{\text{green}\}$ and $h_{\text{TL}}(\mathit{red}) \triangleq \{\text{red}\}$. The resulting Kripke Model would be $\mathcal{M}_{\text{TL}} = (\mathcal{F}_{\text{TL}}, h_{\text{TL}})$.

The Kripke frame and model structures described above have only one accessibility relation R. However, sometimes we need to express multiple modalities within one model (e.g. temporal evolution and agents' knowledge, see Subsection 2.2.6). We can achieve this by augmenting the tuples with the corresponding accessibility relations as follows:

Definition 2.3 (Kripke Frames and Models with Multiple Modalities)**.** Let W be a non-empty set of worlds, $R_1, \ldots, R_n \subseteq W \times W$ binary *accessibility relations*, AP a finite non-empty set of atomic propositions, and $h : AP \to 2^W$ an *assignment.* Then $\mathcal{F}^+ = (W, R_1, \ldots, R_n)$ is a *Kripke frame with multiple modalities* and $\mathcal{M}^+ = (\mathcal{F}^+, h)$ a *Kripke model with multiple modalities.*

Although Kripke frames/models have no concept of local states, which are typically used to represent agents' knowledge (see Subsection 2.1.3), it is possible to express epistemic accessibility (see Subsection 2.2.6 for an introduction to epistemic modalities) explictly using a family of binary relations $\sim_i \subseteq W \times W$ on the set of worlds W for all agents $i \in \mathit{Agt}$. Intuitively, $w_1 \sim_i w_2$ holds iff agent $i \in \mathit{Agt}$ cannot distinguish between worlds $w_1, w_2 \in W$. We can then define a Kripke frame and model with epistemic modalities as:

$$\mathcal{F}_{\mathsf{K}} \triangleq \Big(W, R, (\sim_i)_{i \in \mathit{Agt}}\Big) \qquad\qquad \mathcal{M}_{\mathsf{K}} \triangleq (\mathcal{F}_{\mathsf{K}}, h)$$

Note that even with multiple modalities, Kripke models are still strictly less expressive than concurrent game structures and interpreted systems because they lack the concept of agents' actions.

2.1.2 Concurrent Game Structures

Another framework for modelling systems are concurrent game structures [18], which generalise Kripke semantics by introducing the concept of actions. The temporal evolution of a concurrent game structure depends on the *concurrent actions* of all agents.

Definition 2.4 (Concurrent Game Structures)**.** Let Agt and AP be non-empty sets of agents and atomic propositions respectively. Then a *concurrent game structure* is a tuple $\mathcal{G} \triangleq \langle AP, \mathit{Agt}, \mathit{Act}, \mathit{St}, \lambda, \tau, s_0 \rangle$ where:

- Act is a finite non-empty set of *actions.* $\mathit{Dc} \triangleq \mathit{Agt} \to \mathit{Act}$ is the set of *decisions* which map each agent to its choice of an action.
- St is a finite non-empty set of *states.*
- $\lambda : \mathit{St} \to 2^{AP}$ is a *labelling function* that maps each state to the set of atomic propositions true in that state.
- $\tau : \mathit{St} \times \mathit{Dc} \to \mathit{St}$ is a *transition function.* It maps the current state and a decision to the next state.
- $s_0 \in \mathit{St}$ is an *initial state.*

As we mentioned already, the main difference between Kripke models and concurrent game structures are actions, which enable us to reason about agents' strategies. Apart from them, the frameworks are equivalent[1]: They both have a set of states (W, St), a transition relation/function (R, τ), and a labelling function[2] (h, λ). Similarly to Kripke models, epistemic accessibility can be represented explicitly by a family of binary relations $\sim_i \subseteq St \times St$ for each agent $i \in Agt$.

2.1.3 Interpreted Systems

Interpreted systems are an extension of Kripke semantics for multi-agent systems [65] in which the global states of the whole system are composed of the internal states of all agents.

Definition 2.5 (Interpreted Systems)**.** Let $\Sigma = \{1, \ldots, n\}$ be a set of *players*, E a special agent representing the *environment*, $Agt = \Sigma \cup \{\mathrm{E}\}$ the set of all *agents*, and AP a finite non-empty set of *atomic propositions*. Then an *interpreted system* is a tuple $\mathcal{I} \triangleq \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ where:

- L_i is a finite non-empty set of *internal states* of an agent $i \in Agt$. A tuple $g = (l_1, \ldots, l_n, l_\mathrm{E}) \in L_1 \times \cdots \times L_n \times L_\mathrm{E}$ is called a *global state*. Furthermore, $L_{\sigma\mathrm{E}} \triangleq L_\sigma \times L_\mathrm{E}$ is the set of *local states* of a player $\sigma \in \Sigma$. The set of local states of the environment is equal to its set of internal states, i.e. $L_{\mathrm{EE}} \triangleq L_\mathrm{E}$. $l_i(g)$ and $l_{i\mathrm{E}}(g)$ denote the internal and local state of agent $i \in Agt$ in a global state $g \in G$ respectively.

- Act_i is a finite non-empty set of *actions* that an agent $i \in Agt$ may perform. $Act \triangleq Act_1 \times \cdots \times Act_n \times Act_\mathrm{E}$ denotes the set of *joint actions* of all agents. $a_i(a)$ denotes the action of agent $i \in Agt$ in the joint action $a \in Act$. $Act_A \triangleq \bigcap_{i \in A} Act_i$ is the set of *shared actions* of agents $A \subseteq Agt$ with $A \neq \emptyset$.

- $P_i : L_{i\mathrm{E}} \to 2^{Act_i}$ is the *protocol* of an agent $i \in Agt$. For every local state $l_{i\mathrm{E}} \in L_{i\mathrm{E}}$ of the agent, $P_i(l_{i\mathrm{E}}) \neq \emptyset$ is the set of actions available to agent i. The *global protocol* $P : G \to 2^{Act}$ is defined as $P(g) \triangleq \left\{ a \in Act \;\middle|\; \bigwedge_{i \in Agt} a_i(a) \in P_i(l_{i\mathrm{E}}(g)) \right\}$ for all global states $g \in G$.

- $t_i : L_{i\mathrm{E}} \times Act \to L_i$ is the *evolution function* of the internal state of an agent $i \in Agt$.

- $I \subseteq L_1 \times \cdots \times L_n \times L_\mathrm{E}$ is a finite non-empty set of *initial* global states. $G \subseteq L_1 \times \cdots \times L_n \times L_\mathrm{E}$ is the set of *reachable* global states obtained by considering all the possible evolutions of the system from I.

- $h : AP \to 2^G$ is a *valuation function* that maps each propositional variable to the set of global reachable states in which it is true.

The evolution of the system is described by a *global evolution function* $t : G \times Act \to G$. It is defined as $t(g, a) = g'$ iff $\forall i \in Agt.\, t_i(l_{i\mathrm{E}}(g), a) = l_i(g')$.

Intuitively, the environment agent E represents the part of the model that all agents can "see". That is why its internal states $l_\mathrm{E} \in L_\mathrm{E}$ are included in the other agents' local states $(l_i, l_\mathrm{E}) \in L_{i\mathrm{E}}$. Note that the distinction between internal and local states of agents is often not emphasised in the literature. Internal states are sometimes referred to as *private local states* [65].

Interpreted systems are better suited for epistemic modalities expressing agents' knowledge (see Subsection 2.2.6) because they distinguish between local and global states (unlike Kripke models and concurrent game structures). The relationship between interpreted systems and concurrent game structures is discussed in [65].

We will consider interpreted systems exclusively in the rest of this report because they best represent multi-agent systems (for the reasons outlined above), whose verification our project focuses on.

[1]The explicit initial state s_0 does not increase the expressiveness of concurrent game structures. We could simulate it in Kripke semantics by introducing a new atom $init \in AP$ and setting $h(init) \triangleq \{s_0\}$.

[2]While the signatures of $h : AP \to 2^W$ and $\lambda : St \to 2^{AP}$ differ, they are equivalent (assuming $St = W$) since $h(p) = \lambda^{-1}(p) = \{s \in St \mid p \in \lambda(s)\}$ and $\lambda(s) = h^{-1}(s) = \{p \in AP \mid s \in h(p)\}$ where $p \in AP$ and $s \in St$.

2.2 Specification Languages

Given a property P, we aim to describe it as a formula φ_P. We discuss multiple logics of increasing expressiveness for specifying φ_P. Unfortunately, higher expressiveness is usually accompanied by worse model checking complexity as we shall see in Subsection 2.2.7.

In order to compare the expressiveness of the logics described in this section, we will express the following sample properties using each of them (if possible):

1. A socket is opened and it will stay opened in the next two states.
2. If a device is currently connected, it *can* be disconnected in the future.
3. A task will be finished at some point in the future (no matter what happens).
4. It is possible to permanently delete a file (at some point in the future).
5. If the light is infinitely often red, it will also be infinitely often green (in the same evolution).
6. If a car *can* turn infinitely often left, it *can* also turn infinitely often right (possibly in a different evolution).
7. A firewall and an antivirus can (together) ensure that a computer will never be hacked.
8. If both players play the same strategies, neither of them will ever win.

2.2.1 Linear Temporal Logic

Linear Temporal Logic (LTL) [80] is a temporal logic which models time as a sequence of states, extending infinitely into the future [49]. These possible sequences of states are referred to as *paths*.

Definition 2.6 (Paths and Tracks)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system. A *path* $\pi \in G^\omega$ in $\mathcal{I}$ is an infinite sequence of global states $g_0, g_1, g_2, \ldots \in G$ such that, $\forall i \geq 0 \, \exists a \in Act.\, g_{i+1} = t(g_i, a)$. We write $\pi_{\geq i}$ for the suffix of π starting at g_i, and $\pi(i) = g_i$. $\mathsf{path}(g)$ denotes the set of all paths starting at a global state $g \in G$. $Pth \triangleq \bigcup_{g \in G} \mathsf{path}(g)$ is the set of all paths in $\mathcal{I}$.

A *track* $\tau \in G^+$ in $\mathcal{I}$ is a non-empty finite sequence of global states $g_0, g_1, \ldots, g_{n-1} \in G$ such that, $\forall i \in [0 \,..\, n-2] \, \exists a \in Act.\, g_{i+1} = t(g_i, a)$. The *length* of τ is $|\tau| \triangleq n$. $\mathsf{last}(\tau) \triangleq g_{n-1}$ refers to the *last global state* on τ. $\mathsf{track}(g)$ denotes the set of all tracks starting at a global state $g \in G$. $Trk \triangleq \bigcup_{g \in G} \mathsf{track}(g)$ is the set of all tracks in $\mathcal{I}$.

Intuitively, a path represents a possible evolution of an interpreted system from a given global state $g \in G$ (not necessarily in I). In order to express properties over the possible evolutions of a system, LTL augments propositional logic with temporal connectives[3] X, F, G, U, W, and R.

Definition 2.7 (LTL Syntax)**.** LTL *formulas* are built inductively from the set of atomic propositions AP, by using the following grammar, where $p \in AP$:

$$\varphi ::= p \mid \top \mid \bot \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \mid \varphi \leftrightarrow \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \varphi\,\mathsf{W}\,\varphi \mid \varphi\,\mathsf{R}\,\varphi$$

LTL denotes the infinite set of formulas generated by the above rules.

The syntax allows for the construction of ambiguous formulas like $\mathsf{X}\,a\,\mathsf{U}\,b$, which can be interpreted as either $(\mathsf{X}\,a)\,\mathsf{U}\,b$, or $\mathsf{X}\,(a\,\mathsf{U}\,b)$. In order to resolve such ambiguities, the following binding priorities are used [49]: Unary operators ($\neg$, X, F, G) bind most tightly. The (decreasing) order of the other operators is U, R, W, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$. Hence, the original formula is interpreted as $(\mathsf{X}\,a)\,\mathsf{U}\,b$ because X binds more tightly than U. These rules allow us to use fewer brackets. Note that we do not specify the associativity of binary operators. Thus, the formula $a\,\mathsf{U}\,b\,\mathsf{U}\,c$ is not well-formed, i.e. it needs brackets to disambiguate between $a\,\mathsf{U}\,(b\,\mathsf{U}\,c)$ and $(a\,\mathsf{U}\,b)\,\mathsf{U}\,c$. We shall use similar rules for other logics.

The semantics of LTL are traditionally defined on Kripke models (see Definition 2.2). We will however define them on interpreted systems (see Definition 2.5) here for consistency. Note that, in order to save space, we shall assume throughout the rest of this report that AP is a finite set of propositional atoms.

[3] G, F, and X are sometimes written symbolically as $\square$, $\Diamond$, and $\bigcirc$ respectively.

Definition 2.8 (Ltl Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, and $\pi \in Pth$ a path in $\mathcal{I}$. We define $\mathcal{I}, \pi \models_{\mathsf{LTL}} \varphi$ by induction on $\varphi \in LTL$:

- $\mathcal{I}, \pi \models_{\text{Ltl}} \top$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} p$ iff $\pi(0) \in h(p)$ for $p \in AP$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \neg\varphi$ iff $\mathcal{I}, \pi \not\models_{\text{Ltl}} \varphi$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \wedge \varphi_2$ iff $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1$ and $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_2$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \vee \varphi_2$ iff $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1$ or $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_2$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \rightarrow \varphi_2$ iff $\mathcal{I}, \pi \not\models_{\text{Ltl}} \varphi_1$ or $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_2$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \leftrightarrow \varphi_2$ iff either $\mathcal{I}, \pi \not\models_{\text{Ltl}} \varphi_1$ and $\mathcal{I}, \pi \not\models_{\text{Ltl}} \varphi_2$, or $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1$ and $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_2$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \mathsf{X}\,\varphi$ iff $\mathcal{I}, \pi_{\geq 1} \models_{\text{Ltl}} \varphi$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \mathsf{F}\,\varphi$ iff there exists some $i \geq 0$ such that $\mathcal{I}, \pi_{\geq i} \models_{\text{Ltl}} \varphi$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \mathsf{G}\,\varphi$ iff for all $i \geq 0$ we have $\mathcal{I}, \pi_{\geq i} \models_{\text{Ltl}} \varphi$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there exists some $i \geq 0$ such that $\mathcal{I}, \pi_{\geq i} \models_{\text{Ltl}} \varphi_2$ and for all numbers $0 \leq j < i$ we have $\mathcal{I}, \pi_{\geq j} \models_{\text{Ltl}} \varphi_1$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \,\mathsf{W}\, \varphi_2$ iff either *(i)* there exists some $i \geq 0$ such that $\mathcal{I}, \pi_{\geq i} \models_{\text{Ltl}} \varphi_2$ and for all $0 \leq j < i$ we have $\mathcal{I}, \pi_{\geq j} \models_{\text{Ltl}} \varphi_1$, or *(ii)* for all $k \geq 0$ we have $\mathcal{I}, \pi_{\geq k} \models_{\text{Ltl}} \varphi_1$;
- $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi_1 \,\mathsf{R}\, \varphi_2$ iff either *(i)* there exists some $i \geq 0$ such that $\mathcal{I}, \pi_{\geq i} \models_{\text{Ltl}} \varphi_1$ and for all $0 \leq j \leq i$ we have $\mathcal{I}, \pi_{\geq j} \models_{\text{Ltl}} \varphi_2$, or *(ii)* for all $k \geq 0$, we have $\mathcal{I}, \pi_{\geq k} \models_{\text{Ltl}} \varphi_2$.

A formula $\varphi \in LTL$ is true *at a global state* $g \in G$ in an interpreted system $\mathcal{I}$ (written as $\mathcal{I}, g \models_{\text{Ltl}} \varphi$) iff it is true *on all paths* $\pi \in \mathsf{path}(g)$ starting at g (i.e. $\mathcal{I}, \pi \models_{\text{Ltl}} \varphi$).

The intuitive meanings of the temporal operators X, F, G, U, W, and R are "neXt state", "some Future state", "all future states (Globally)", "Until", "Weak-until", and "Release" respectively [49]. For example, the Ltl formula $a \,\mathsf{U}\, b$ expresses the property that a holds *until* b holds. There also exist past temporal connectives Y, S, O, and H which are opposite to X, U, F, and G respectively [49]. Their intuitive meanings are "Yesterday", "Since", "Once", and "Historically". However, adding these operators does *not* increase the expressiveness of Ltl.

From this point onwards, we will omit the operators $\bot$, $\rightarrow$, and $\leftrightarrow$ in all logics for the sake of conciseness. We can always express them using the following propositional equivalences:

$$\bot \equiv \neg\top \qquad \varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2 \qquad \varphi_1 \leftrightarrow \varphi_2 \equiv (\varphi_1 \wedge \varphi_2) \vee (\neg\varphi_1 \wedge \neg\varphi_2)$$

In fact, the set $\{\top, \neg, \wedge, \mathsf{X}, \mathsf{U}\}$ forms an *adequate set of connectives*[4] for Ltl [49], i.e. all the other operators ($\vee$, F, G, W, and R) can be expressed using equivalences:

$$\varphi_1 \vee \varphi_2 \equiv \neg(\varphi_1 \wedge \varphi_2) \qquad \mathsf{F}\,\varphi \equiv \top \,\mathsf{U}\, \varphi \qquad \mathsf{G}\,\varphi \equiv \neg\mathsf{F}\,\neg\varphi \equiv \neg(\top \,\mathsf{U}\, \neg\varphi)$$
$$\varphi_1 \,\mathsf{W}\, \varphi_2 \equiv \varphi_1 \,\mathsf{U}\, \varphi_2 \vee \mathsf{G}\,\varphi_1 \equiv \varphi_1 \,\mathsf{U}\, \varphi_2 \vee \neg(\top \,\mathsf{U}\, \neg\varphi_1) \qquad \varphi_1 \,\mathsf{R}\, \varphi_2 \equiv \neg(\neg\varphi_1 \,\mathsf{U}\, \neg\varphi_2)$$

Model checking of an Ltl formula φ is typically performed by constructing a non-deterministic Büchi automaton $\mathfrak{A}_{\neg\varphi} = (S_{\mathfrak{A}}, 2^{AP}, I_{\mathfrak{A}}, R_{\mathfrak{A}}, \mathcal{F})$ equivalent to $\neg\varphi$ (see Section 2.4) and then checking if no fair path exists in the product of the interpreted system and the automaton. More precisely, $\mathcal{I}, g \not\models_{\text{Ltl}} \varphi$ iff there exists an initial state $s_I \in I_{\mathfrak{A}}$ of the automaton and a path $\pi \in \mathsf{path}((g, s_I))$ in $\mathcal{I} \times \mathfrak{A}_{\neg\varphi}$ starting at (g, s_I) such that $\mathcal{F}$ is visited infinitely often along π. Please refer to [34] for more details about how this is performed in practice by converting Ltl model checking to Ctl model checking with fairness

[4]Note that an adequate set is *not* necessarily unique for a given logic. For example, we could replace $\wedge$ with $\vee$ and/or U with R and the result would still be an adequate set for Ltl.

conditions using the tableau method. Another popular approach to LTL model checking is *bounded model checking* [32], which transforms the model checking problem into a propositional satisfiability problem.

We shall now demonstrate the expressive power of LTL. The sample properties listed at the beginning of this section can be expressed in LTL as follows:

1. $\mathit{open} \wedge \mathsf{X}\,\mathit{open} \wedge \mathsf{X}\,\mathsf{X}\,\mathit{open}$.

2. This property cannot be expressed directly in LTL because it requires existential quantification (while LTL formulas are implicitly universally quantified over *all* paths). However, it is possible to express its *converse* as $\mathit{connected} \wedge \mathsf{G}\,\mathit{connected}$. If the converse does *not* hold in a state, then the property is satisfied. Note that this shows that $\mathcal{I}, g \models_{\text{LTL}} \neg\varphi$ is in general *not equivalent* to $\mathcal{I}, g \not\models_{\text{LTL}}$ for a global state $g \in G$. Compare this with paths $\pi \in \mathit{Pth}$ for which $\mathcal{I}, \pi \models_{\text{LTL}} \neg\varphi$ is *defined* as $\mathcal{I}, \pi \not\models_{\text{LTL}} \varphi$.

3. $\mathsf{F}\,\mathit{finished}$.

4, 6. Neither the properties nor their converses can be expressed in LTL because they combine both existential and universal quantification[5].

5 $\mathsf{G}\,\mathsf{F}\,\mathit{red} \rightarrow \mathsf{G}\,\mathsf{F}\,\mathit{green}$.

7–8. These properties cannot be expressed in LTL because it cannot refer to agents' strategies.

We can see that LTL can only express properties that should hold on *all* paths. While it is possible to assert the existence of a path (see item 2 above), existential and universal path quantification cannot be combined within a single LTL formula (see items 4 and 6). Certain important properties therefore cannot be expressed in LTL. This shortcoming is addressed by Computation Tree Logic.

2.2.2 Computation Tree Logic

Computation Tree Logic (CTL) [33] is a temporal logic which allows us to quantify over paths explicitly [49]. Compared to LTL, CTL adds quantifiers A and E expressing "for all paths" and "there exists a path" respectively. Again, CTL extends the syntax of propositional logic with path and temporal connectives:

Definition 2.9 (CTL Syntax)**.** CTL *formulas* are built inductively from the set of atomic propositions AP, by using the following grammar, where $p \in AP$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AX}\,\varphi \mid \mathsf{EX}\,\varphi \mid \mathsf{AF}\,\varphi \mid \mathsf{EF}\,\varphi \mid \mathsf{AG}\,\varphi \mid \mathsf{EG}\,\varphi \mid \mathsf{A}[\varphi\,\mathsf{U}\,\varphi] \mid \mathsf{E}[\varphi\,\mathsf{U}\,\varphi]$$

CTL denotes the infinite set of formulas generated by the above rules.

Observe that every path quantifier is coupled with a temporal operator and vice versa. The binding priorities of the operators are the same as for LTL (e.g. $\mathsf{EF}\,a \wedge b$ is interpreted as $(\mathsf{EF}\,a) \wedge b$). Similarly to LTL, we define CTL semantics on interpreted systems (see Definition 2.5) for consistency:

Definition 2.10 (CTL Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, and $g \in G$ a global state. We define $\mathcal{I}, g \models_{\text{CTL}} \varphi$ by induction on $\varphi \in LTL$:

- $\mathcal{I}, g \models_{\text{CTL}} \top$;
- $\mathcal{I}, g \models_{\text{CTL}} p$ iff $g \in h(p)$ for $p \in AP$;
- $\mathcal{I}, g \models_{\text{CTL}} \neg\varphi$ iff $\mathcal{I}, g \not\models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \varphi_1 \wedge \varphi_2$ iff $\mathcal{I}, g \models_{\text{CTL}} \varphi_1$ and $\mathcal{I}, g \models_{\text{CTL}} \varphi_2$;
- $\mathcal{I}, g \models_{\text{CTL}} \varphi_1 \vee \varphi_2$ iff $\mathcal{I}, g \models_{\text{CTL}} \varphi_1$ or $\mathcal{I}, g \models_{\text{CTL}} \varphi_2$;

[5] This might not be immediately obvious in property 6. Recall that $\varphi_1 \rightarrow \varphi_2 \equiv \neg\varphi_1 \vee \varphi_2$. Hence the property can be interpreted as: on all paths the car does not turn infinitely often left or there is a path on which the car turns right infinitely often.

- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{AX}\,\varphi$ iff for all paths $\pi \in \mathsf{path}(g)$, we have $\mathcal{I}, \pi(1) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{EX}\,\varphi$ iff there exists a path $\pi \in \mathsf{path}(g)$ such that $\mathcal{I}, \pi(1) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{AF}\,\varphi$ iff for all paths $\pi \in \mathsf{path}(g)$ there is some $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{EF}\,\varphi$ iff there exist a path $\pi \in \mathsf{path}(g)$ and $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{AG}\,\varphi$ iff for all paths $\pi \in \mathsf{path}(g)$ and $i \geq 0$ we have $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{EG}\,\varphi$ iff there exists a path $\pi \in \mathsf{path}(g)$ such that for all $i \geq 0$ we have $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi$;
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{A}[\varphi_1 \,\mathsf{U}\, \varphi_2]$ iff for all paths $\pi \in \mathsf{path}(g)$ there exists $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi_2$, and for all $0 \leq j < i$, $\mathcal{I}, \pi(j) \models_{\text{CTL}} \varphi_1$.
- $\mathcal{I}, g \models_{\text{CTL}} \mathsf{E}[\varphi_1 \,\mathsf{U}\, \varphi_2]$ iff there exists a path $\pi \in \mathsf{path}(g)$ and $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{CTL}} \varphi_2$, and for all $0 \leq j < i$, $\mathcal{I}, \pi(j) \models_{\text{CTL}} \varphi_1$.

The intuitive meanings of the temporal operators are the same as for Ltl (see Subsection 2.2.1). For example, $\mathsf{EG}\,p$ means that there exists an infinite path (starting in the current global state) such that p holds at every single state of the path. One possible adequate set of connectives for Ctl is $\{\top, \neg, \wedge, \mathsf{EX}, \mathsf{AF}, \mathsf{EU}\}$ [49].

Model checking of a Ctl formula φ is typically[6] performed by recursively calculating the sets of states $\|\psi\|_{\mathcal{I}} \triangleq \{g \in G \mid \mathcal{I}, g \models_{\text{CTL}} \psi\}$ of the interpreted system $\mathcal{I}$ at which subformulas ψ of φ hold in a bottom-up manner. For example, assuming that we have already calculated $\|\psi\|_{\mathcal{I}}$, $\|\mathsf{EX}\,\psi\|_{\mathcal{I}}$ is simply the set of global states of $\mathcal{I}$ which have a possible transition into $\|\psi\|_{\mathcal{I}}$, which we denote $\mathsf{pre}_{\exists}(\|\psi\|_{\mathcal{I}})$.

Definition 2.11 (Predecessors)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system and $X \subseteq G$ a set of global states. The *existential* and *universal predecessors* of X are defined as $\mathsf{pre}_{\exists}(X) \triangleq \{g \in G \mid \exists a \in P(g).\, t(g,a) \in X\}$ and $\mathsf{pre}_{\forall}(X) \triangleq \{g \in G \mid \forall a \in P(g).\, t(g,a) \in X\}$ respectively.

Once we have calculated $\|\varphi\|_{\mathcal{I}}$, determining whether the formula φ holds at a given global state $g \in G$ (i.e. $\mathcal{I}, g \models_{\text{CTL}} \varphi$) is equivalent to checking whether $g \in \|\varphi\|_{\mathcal{I}}$. The recursive model checking algorithm $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\cdot)$ for Ctl, which calculates $\|\varphi\|_{\mathcal{I}}$, is defined as follows [49]:

Definition 2.12 (Ctl Model Checking Algorithm)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system and $\varphi \in \mathit{CTL}$ a Ctl formula. The function $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}} : \mathit{CTL} \to 2^G$ is defined inductively as follows:

- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\top) \triangleq G$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(p) \triangleq h(p)$ for $p \in AP$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\neg\varphi) \triangleq G \setminus \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_1 \wedge \varphi_2) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_1) \cap \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_2)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_1 \vee \varphi_2) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_1) \cup \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_2)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{AX}\,\varphi) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\neg\mathsf{EX}\,\neg\varphi)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{EX}\,\varphi) \triangleq \mathsf{pre}_{\exists}\left(\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi)\right)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{AF}\,\varphi) \triangleq \mathrm{lfp}_X\left[\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi) \cup \mathsf{pre}_{\forall}(X)\right]$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{EF}\,\varphi) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{E}[\top \,\mathsf{U}\, \varphi])$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{AG}\,\varphi) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\neg\mathsf{EF}\,\neg\varphi)$;

[6]Alternative approaches exist. For example, a model checking algorithm for ACtl (a universal fragment of Ctl where all path quantifiers are A) based on *bounded model checking* is presented in [87].

- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{EG}\,\varphi) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\neg\mathsf{AF}\,\neg\varphi)$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{A}[\varphi_1\,\mathsf{U}\,\varphi_2]) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\neg\,(\mathsf{E}[\neg\varphi_2\,\mathsf{U}\,(\neg\varphi_1 \wedge \neg\varphi_2)] \vee \mathsf{EG}\,\neg\varphi_2))$;
- $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\mathsf{E}[\varphi_1\,\mathsf{U}\,\varphi_2]) \triangleq \mathrm{lfp}_X\left[\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_2) \cup \left(\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi_1) \cap \mathsf{pre}_{\exists}\,(X)\right)\right]$;

where lfp denotes the least fixed point.

Given a CTL formula φ, the algorithm calculates the set of global states of $\mathcal{I}$ at which it holds. This fact is stated formally in the following proposition.

Proposition 2.1. Let $\mathcal{I}$ be an interpreted system and $\varphi \in \mathit{CTL}$ an arbitrary CTL formula. Then we have $\mathsf{SAT}^{\mathcal{I}}_{\text{CTL}}(\varphi) = \|\varphi\|_{\mathcal{I}} \triangleq \{g \in G \mid \mathcal{I}, g \models_{\text{CTL}} \varphi\}$.

Similarly to LTL, we now demonstrate the expressive power of CTL. The sample properties listed at the beginning of this section can be expressed in CTL as follows:

1. $\mathit{open} \wedge \mathsf{AX}\,\mathit{open} \wedge \mathsf{AX}\,\mathsf{AX}\,\mathit{open}$.
2. $\mathit{connected} \rightarrow \mathsf{EF}\,\neg\mathit{connected}$.
3. $\mathsf{AF}\,\mathit{finished}$.
4. $\mathsf{EF}\,\mathsf{AG}\,\mathit{deleted}$.

6–8. These properties cannot be expressed in CTL because it does not refer to paths (properties 5 and 6[7]) and agents' strategies (properties 7 and 8).

Unlike LTL, CTL can express properties which combine existential and universal path quantifiers (property 4). However, it is limited by the fact that every temporal operator must be coupled with a path quantifier, which prevents CTL from expressing path specifications like property 5. Intuitively, we would like to express it as $\mathsf{A}[\mathsf{G}\,\mathsf{F}\,\mathit{red} \rightarrow \mathsf{G}\,\mathsf{F}\,\mathit{green}]$, which is exactly what Full Branching Time Logic allows us to do.

2.2.3 Full Branching Time Logic

Full Branching Time Logic (CTL*) [37] is a superset of both LTL and CTL [49]. Moreover, it is strictly more expressive than the union of LTL and CTL, i.e. there exist CTL* formulas which cannot be expressed in either of the two logics. CTL* supports CTL path quantifiers A and E but relaxes the restrictions on their coupling with temporal operators:

Definition 2.13 (CTL* Syntax)**.** CTL* *formulas* are built inductively from the set of atomic propositions AP. State formulas φ and path formulas ψ are defined by the following mutually recursive grammar, where $p \in AP$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{A}[\psi] \mid \mathsf{E}[\psi]$$
$$\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\,\psi \mid \mathsf{F}\,\psi \mid \mathsf{G}\,\psi \mid \psi\,\mathsf{U}\,\psi$$

CTL^* denotes the infinite set of formulas generated by the above rules.

CTL* semantics are very similar to LTL and CTL semantics with only a few adjustments. We will also define them on interpreted systems (see Definition 2.5) for consistency:

Definition 2.14 (CTL* Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, \mathit{Act}_i, P_i, t_i)_{i \in \mathit{Agt}}, I, h \right\rangle$ be an interpreted system, $g \in G$ a global state, and $\pi \in \mathit{Pth}$ a path in $\mathcal{I}$. We define $\mathcal{I}, g \models_{\text{CTL}^*} \varphi$ and $\mathcal{I}, \pi \models_{\text{CTL}^*} \psi$ by induction on state formulas φ and path formulas ψ:

[7]It would appear that the CTL formula $\mathsf{EG}\,\mathsf{EF}\,\varphi$ means that there exists a path along which φ holds infinitely often. Unfortunately, that is not the case. Consider a simple interpreted system with three global states g_1, g_2, g_3, four possible transitions $g_1 \rightarrow g_1$, $g_1 \rightarrow g_2$, $g_2 \rightarrow g_3$, and $g_3 \rightarrow g_3$, and an assignment $h(p) \triangleq \{g_2\}$. Then we have $\mathcal{I}, g_2 \models_{\text{CTL}} p$, so $\mathcal{I}, g_1 \models_{\text{CTL}} \mathsf{EF}\,p$ (g_1 is an existential predecessor of g_2) and $\mathcal{I}, g_1 \models_{\text{CTL}} \mathsf{EG}\,\mathsf{EF}\,p$ (cycle in g_1). However, there is no path starting in g_1 along which p would be infinitely often true. A similar relationship between the CTL formula $\mathsf{AG}\,\mathsf{AF}\,p$ and the LTL formula $\mathsf{G}\,\mathsf{F}\,p$ is shown in [49].

- $\mathcal{I}, g \models_{\text{CTL}^*} \mathsf{A}[\psi]$ iff for all paths $\pi \in \mathsf{path}(g)$ starting at g we have $\mathcal{I}, \pi \models_{\text{CTL}^*} \psi$;
- $\mathcal{I}, g \models_{\text{CTL}^*} \mathsf{E}[\psi]$ iff there exists a path $\pi \in \mathsf{path}(g)$ starting at g such that $\mathcal{I}, \pi \models_{\text{CTL}^*} \psi$;
- $\mathcal{I}, g \models_{\text{CTL}^*} \varphi$ is defined in the same way as $\mathcal{I}, g \models_{\text{CTL}} \varphi$ for other CTL* state formulas;
- $\mathcal{I}, \pi \models_{\text{CTL}^*} \varphi$ iff $\mathcal{I}, \pi(0) \models_{\text{CTL}^*} \varphi$;
- $\mathcal{I}, \pi \models_{\text{CTL}^*} \psi$ is defined in the same way as $\mathcal{I}, \pi \models_{\text{LTL}} \psi$ for other CTL* path formulas.

Since LTL and CTL are subsets of CTL*, any LTL/CTL formula can be transformed into an equivalent CTL* formula:

- Let $\psi \in LTL$ be an LTL formula. It is equivalent to the CTL* formula $\mathsf{A}[\psi]$. This can be seen from the semantics of LTL:

$$\begin{aligned} \mathcal{I}, g \models_{\text{LTL}} \psi \quad & \text{iff} \quad \forall \pi \in \mathsf{path}(g). \mathcal{I}, \pi \models_{\text{LTL}} \psi \\ & \text{iff} \quad \forall \pi \in \mathsf{path}(g). \mathcal{I}, \pi \models_{\text{CTL}^*} \psi \\ & \text{iff} \quad \mathcal{I}, g \models_{\text{CTL}^*} \mathsf{A}[\psi] \end{aligned}$$

  where the second equivalence follows from the fact that ψ contains no path quantifiers.

- CTL is a subset of CTL* in which *(i)* we do not allow Boolean combinations of path formulas, and *(ii)* each temporal connective must be preceeded by a path quantifier. An alternative definition of CTL syntax can thus be obtained by restricting path formulas in CTL* syntax (Definition 2.13):

$$\psi ::= \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

  Hence, any CTL formula is also an equivalent CTL* formula.

CTL* model checking can be performed in the following bottom-up manner[8]: In order to calculate $\|\mathsf{A}[\varphi]\|$, we replace each subformula $\mathsf{A}[\psi]$ of φ with a new atom $p \in AP$ such that $h(p) \triangleq \|\mathsf{A}[\psi]\|_{\mathcal{I}}$, which we calculate recursively. φ is now an LTL formula, so we can use the method involving automata outlined in Subsection 2.2.1 (or any other method for model checking LTL) to calculate $\|\mathsf{A}[\varphi]\|_{\mathcal{I}}$. Please refer to [39] for more details about this reduction from CTL* to LTL model checking.

Let us now investigate the expressive power of CTL*. The sample properties listed at the beginning of this section can be expressed in CTL* as follows:

1. $\mathsf{A}[\mathit{open} \wedge \mathsf{X}\,\mathit{open} \wedge \mathsf{X}\,\mathsf{X}\,\mathit{open}]$.
2. $\mathit{connected} \rightarrow \mathsf{E}[\mathsf{F}\,\neg \mathit{connected}]$.
3. $\mathsf{A}[\mathsf{F}\,\mathit{finished}]$.
4. $\mathsf{E}[\mathsf{F}\,\mathsf{A}[\mathsf{G}\,\mathit{deleted}]]$.
5. $\mathsf{A}[\mathsf{G}\,\mathsf{F}\,\mathit{red} \rightarrow \mathsf{G}\,\mathsf{F}\,\mathit{green}]$.
6. $\mathsf{E}[\mathsf{G}\,\mathsf{F}\,\mathit{left}] \rightarrow \mathsf{E}[\mathsf{G}\,\mathsf{F}\,\mathit{right}]$.

7–8. These properties cannot be expressed in CTL* because it does not refer to agents' strategies.

We can see that CTL* can express all properties supported by LTL (1, 3, 5) and CTL (1–4). Moreover, it can express property 6, which could not be represented as a LTL or CTL formula. The only specifications (on our list) which CTL* cannot express are those that involve reasoning about agents' ability to enforce certain properties on the system. This issue is addressed by Alternating-Time Temporal Logic.

[8] To make things simpler, we only consider the universal path quantifier A. The existential quantifier E can be equivalently expressed as $\neg\mathsf{A}\neg$.

2.2.4 Alternating-Time Temporal Logic

Alternating-Time Temporal Logic (ATL) [18] is an extension of CTL for multi-agent systems. Instead of universal and existential path quantification, ATL allows for a more fine-grained control by parametrising temporal operators with sets of agents. This makes ATL more suitable for reasoning about open systems (such as multi-agent systems), which interact with their environment [18]. To do this, ATL introduces the $\langle\langle A\rangle\rangle$ quantifier, which ranges over the set of paths that can be enforced by the agents $A \subseteq Agt$ (regardless of what action the other agents $Agt \setminus A$ carry out) and replaces the CTL path quantifiers A and E.

Definition 2.15 (ATL Syntax)**.** ATL *formulas* are built inductively from the set of atomic propositions AP and agents Agt, by using the following grammar, where $p \in AP$ and $A \subseteq Agt$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\langle A\rangle\rangle \mathsf{X}\, \varphi \mid \langle\langle A\rangle\rangle \mathsf{F}\, \varphi \mid \langle\langle A\rangle\rangle \mathsf{G}\, \varphi \mid \langle\langle A\rangle\rangle [\varphi\, \mathsf{U}\, \varphi]$$

ATL denotes the infinite set of formulas generated by the above rules.

Unlike LTL and CTL semantics, ATL semantics are traditionally defined on concurrent game structures (see Definition 2.4). However, we will define them here on interpreted systems (see Definition 2.5) like we did for the previous formalisms for consistency. We first need to define the concept of a *strategy.*

Definition 2.16 (Individual Strategies)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system and $i \in Agt$ an agent. Then a partial function $f_i : Trk \rightharpoonup Act_i$ such that for all $\tau \in \mathrm{dom}(f_i)$, $f_i(\tau) \in P_i(l_{i\mathrm{E}}(\mathsf{last}(\tau)))$, is an *(individual) strategy* for agent i which maps tracks to actions. $Str_i \subseteq Trk \rightharpoonup Act_i$ denotes the set of all strategies for agent i in $\mathcal{I}$. The set of all strategies is defined as $Str \triangleq \bigcup_{i \in Agt} Str_i$.

Let $A \subseteq Agt$ be a set of agents. Then an *agent assignment* is a function $\alpha : A \to Str$ which maps each agent $i \in A$ to a strategy $\alpha(i) \in Str_i$. The set of all agent assignment for agents A is denoted as $AAsg_A$. The set of all agent assignments is defined as $AAsg \triangleq \bigcup_{A \subseteq Agt} AAsg_A$.

Intuitively, an agent assignment maps agents to strategies, which in turn map histories of the system to the next actions of the agents. Given an agent assignment $\alpha \in AAsg_A$ for a group of agents $A \subseteq Agt$, we want to find the set of paths that the agents can enforce (if they follow α) regardless of the other agents' behaviour. This is referred to as the set of *outcomes.*

Definition 2.17 (Outcomes)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $g \in G$ a global state, $A \subseteq Agt$ a set of agents, and $\alpha \in AAsg_A$ an agent assignment for A. The set of *outcomes* $\mathsf{out}(g, \alpha)$ contains all paths $\pi \in Pth$ such that *(i)* $\pi(0) = g$, and *(ii)* for all $k \geq 0$, there exists a joint action $a \in P(\pi(k))$, such that $t(\pi(k), a) = \pi(k+1)$ and for all $i \in A$, if $\pi_{\leq k} \in \mathrm{dom}(\alpha(i))$, then $a_i(a) = \alpha(i)(\pi_{\leq k})$.

Finally, we are ready to define ATL semantics. Intuitively, a formula $\langle\langle A\rangle\rangle\varphi$ is true iff the agents $A \subseteq Agt$ can enforce φ.

Definition 2.18 (ATL Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $g \in G$ a global state, and $A \subseteq Agt$ a set of agents. We define $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi$ by induction on $\varphi \in ATL$:

- $\mathcal{I}, g \models_{\mathrm{ATL}} \top$;
- $\mathcal{I}, g \models_{\mathrm{ATL}} p$ iff $g \in h(p)$ for $p \in AP$;
- $\mathcal{I}, g \models_{\mathrm{ATL}} \neg\varphi$ iff $\mathcal{I}, g \not\models_{\mathrm{ATL}} \varphi$;
- $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_1 \wedge \varphi_2$ iff $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_1$ and $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_2$;
- $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_1 \vee \varphi_2$ iff $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_1$ or $\mathcal{I}, g \models_{\mathrm{ATL}} \varphi_2$;
- $\mathcal{I}, g \models_{\mathrm{ATL}} \langle\langle A\rangle\rangle \mathsf{X}\, \varphi$ iff there exists an agent assignment $\alpha \in AAsg_A$, such that for all paths $\pi \in \mathsf{out}(s, \alpha)$ we have $\mathcal{I}, \pi(1) \models_{\mathrm{ATL}} \varphi$;

- $\mathcal{I}, g \models_{\text{ATL}} \langle\!\langle A \rangle\!\rangle \mathsf{F}\, \varphi$ iff there exists an agent assignment $\alpha \in AAsg_A$, such that for all paths $\pi \in \mathsf{out}(s, \alpha)$ there exists $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{ATL}} \varphi$;
- $\mathcal{I}, g \models_{\text{ATL}} \langle\!\langle A \rangle\!\rangle \mathsf{G}\, \varphi$ iff there exists an agent assignment $\alpha \in AAsg_A$, such that for all paths $\pi \in \mathsf{out}(s, \alpha)$ and $i \geq 0$ we have $\mathcal{I}, \pi(i) \models_{\text{ATL}} \varphi$;
- $\mathcal{I}, g \models_{\text{ATL}} \langle\!\langle A \rangle\!\rangle [\varphi_1 \,\mathsf{U}\, \varphi_2]$ iff there exists an agent assignment $\alpha \in AAsg$, such that for all paths $\pi \in \mathsf{out}(s, \alpha)$ there exists $i \geq 0$ such that $\mathcal{I}, \pi(i) \models_{\text{ATL}} \varphi_2$ and for all $0 \leq j < i$ we have $\mathcal{I}, \pi(j) \models_{\text{ATL}} \varphi_1$.

The meaning of the temporal operators is the same as in Ltl and Ctl. For example, the Atl formula $\langle\!\langle \{a, b\} \rangle\!\rangle \mathsf{G}\, p$ means that there are strategies for agents a and b such that no matter what the other agents do, p will be always true. We can now see the relationship between Ctl and Atl: Given a Ctl formula $\varphi \in CTL$, an equivalent Atl formula $\varphi' \in ATL$ can be obtained by replacing every subformula of the form $\mathsf{E}[\psi]$ and $\mathsf{A}[\psi]$ with $\langle\!\langle Agt \rangle\!\rangle \psi$ and $\langle\!\langle \emptyset \rangle\!\rangle \psi$ respectively [18].

Similarly to Ctl* (see Subsection 2.2.3), which extends Ctl by relaxing its syntax constraints (see Subsection 2.2.2), *Full Alternating-Time Logic* (Atl*) extends Atl by relaxing its syntax constraints, i.e. *(i)* Atl* syntax distinguishes between state and path formulas and *(ii)* quantifiers ($\langle\!\langle A \rangle\!\rangle$) and temporal operators (X, F, G, U) need not be coupled any more. Atl* semantics are modified analogously (see Definition 2.14). The result is a strictly more expressive logic [18].

The model checking algorithm for Atl is very similar to the one for Ctl (see Definition 2.12). The main difference is that the predecessor function must take into account the quantified group of agents. For example, if we have already calculated the set of global states $\|\varphi\|_{\mathcal{I}}$ of an interpreted system $\mathcal{I}$ where the Atl formula φ holds, the set of global states $\|\langle\!\langle A \rangle\!\rangle \mathsf{F}\, \varphi\|_{\mathcal{I}}$ where $\langle\!\langle A \rangle\!\rangle \mathsf{F}\, \varphi$ holds ($A \subseteq Agt$ is a set of agents) is equal to $\mathrm{lfp}_X\ [\|\varphi\|_{\mathcal{I}} \cup \mathsf{pre}_A(X)]$ where:

$$\mathsf{pre}_A(X) \triangleq \left\{ g \in G \;\middle|\; \exists a \in (Act_i)_{i \in A}\, \forall a' \in (Act_i)_{i \in Agt \setminus A} \,.\, t(g, (a, a')) \in X \right\}$$

Please refer to [65] for a complete model checking algorithm for Atl. Model checking Atl* is much more difficult. Although a theoretical model checking algorithm for Atl* is provided in [18] (in the form of a constructive proof of Atl* model checking complexity), we are not aware of any existing tool that would support it. If this is true, then we have created the first model checker for Atl* as part of this project (see Section 1.3).

We shall now demonstrate the expressive power of Atl. The sample properties listed at the beginning of this section can be expressed in Atl as follows:

1. $open \wedge \langle\!\langle \emptyset \rangle\!\rangle \mathsf{X}\, open \wedge \langle\!\langle \emptyset \rangle\!\rangle \mathsf{X} \langle\!\langle \emptyset \rangle\!\rangle \mathsf{X}\, open$.
2. $connected \to \langle\!\langle Agt \rangle\!\rangle \mathsf{F}\, \neg connected$.
3. $\langle\!\langle \emptyset \rangle\!\rangle \mathsf{F}\, finished$.
4. $\langle\!\langle Agt \rangle\!\rangle \mathsf{F} \langle\!\langle \emptyset \rangle\!\rangle \mathsf{G}\, deleted$.
5. This property cannot be expressed in Atl because it does not refer to paths. It can however be expressed in Atl* as $\langle\!\langle \emptyset \rangle\!\rangle (\mathsf{G}\,\mathsf{F}\, red \to \mathsf{G}\,\mathsf{F}\, green)$.
6. This property cannot be expressed in Atl because it does not refer to paths. It can however be expressed in Atl* as $[\langle\!\langle Agt \rangle\!\rangle \mathsf{G}\,\mathsf{F}\, left] \to [\langle\!\langle Agt \rangle\!\rangle \mathsf{G}\,\mathsf{F}\, right]$.
7. $\langle\!\langle \{\text{firewall}, \text{antivirus}\} \rangle\!\rangle \mathsf{G}\, \neg hacked$.
8. This property cannot be expressed in Atl/Atl* because it does not refer to strategies directly.

Atl and Atl* allow us to express almost all properties on our list. The only exception is property 8, which cannot be expressed in either Atl or Atl* because it refers to agents' strategies directly, while both logics treat strategies only implicitly (via the $\langle\!\langle A \rangle\!\rangle$ quantifier). This limitation of Atl and Atl* is addressed by Strategy Logic.

2.2.5 Strategy Logic

Strategy Logic (SL) is a new formalism introduced in [76], which strictly subsumes ATL*. Instead of quantifying over agents in formulas, SL quantifies over strategies explicitly, allowing us to express properties like *"agents a and b share the same strategy"* or even game-theoretic concepts like Nash equilibria. To do this, it augments the syntax of LTL (see Definition 2.7) with three operators: universal strategy quantifier $[\![x]\!]\varphi$, existential strategy quantifier $\langle\!\langle x\rangle\!\rangle\varphi$, and agent binding $(a, x)\varphi$.

Definition 2.19 (SL Syntax)**.** SL *formulas* are built inductively from the set of atomic propositions AP, strategy variables Var, and agents Agt, by using the following grammar, where $p \in AP$, $x \in Var$, and $i \in Agt$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \langle\!\langle x\rangle\!\rangle\varphi \mid [\![x]\!]\varphi \mid (i, x)\varphi$$

SL denotes the infinite set of formulas generated by the above rules.

We define $\mathsf{vars}(\varphi) \subseteq Var$ to be the set of strategy variables quantified in an SL formula φ (e.g. $\mathsf{vars}(\langle\!\langle x\rangle\!\rangle[\![y]\!](a, x)(b, y)\mathsf{X}\langle\!\langle z\rangle\!\rangle(b, z)\mathsf{F}\,p) = \{x, y, z\}$). Similarly to first-order languages, we define the set of *free agents/variables* of a formula which contains *(i)* all agents for which there is no binding after the occurrence of a temporal operator and *(ii)* all variables for which there is a binding but no quantifications.

Definition 2.20 (Free Agents/Variables)**.** The set of *free agents/variables* of an SL formula is given by the function $\mathsf{free} : SL \to 2^{Agt \cup Var}$ defined as follows:

1. $\mathsf{free}(\top) \triangleq \emptyset$;
2. $\mathsf{free}(p) \triangleq \emptyset$, where $p \in AP$;
3. $\mathsf{free}(\neg\varphi) \triangleq \mathsf{free}(\varphi)$;
4. $\mathsf{free}(\varphi_1 \wedge \varphi_2) \triangleq \mathsf{free}(\varphi_1) \cup \mathsf{free}(\varphi_2)$;
5. $\mathsf{free}(\mathsf{Op}\,\varphi) \triangleq Agt \cup \mathsf{free}(\varphi)$, where $\mathsf{Op} \in \{\mathsf{X}, \mathsf{F}, \mathsf{G}\}$;
6. $\mathsf{free}(\varphi_1\,\mathsf{U}\,\varphi_2) \triangleq Agt \cup \mathsf{free}(\varphi_1) \cup \mathsf{free}(\varphi_2)$;
7. $\mathsf{free}(\mathsf{Qn}\,\varphi) \triangleq \mathsf{free}(\varphi) \setminus \{x\}$, where $\mathsf{Qn} \in \{[\![x]\!], \langle\!\langle x\rangle\!\rangle : x \in Var\}$;
8. $\mathsf{free}((i, x)\varphi) \triangleq \mathsf{free}(\varphi)$, if $i \notin \mathsf{free}(\varphi)$, where $i \in Agt$ and $x \in Var$;
9. $\mathsf{free}((i, x)\varphi) \triangleq (\mathsf{free}(\varphi) \setminus \{i\}) \cup \{x\}$, if $i \in \mathsf{free}(\varphi)$, where $i \in Agt$ and $x \in Var$.

A formula $\varphi \in SL$ is *agent-closed* (resp. *variable-closed*) iff $\mathsf{free}(\varphi) \cap Agt = \emptyset$ (resp. $\mathsf{free}(\varphi) \cap Var = \emptyset$). A formula $\varphi \in SL$ is a *sentence* iff it is both agent-closed and variable-closed.

Consider an interpreted system with agents $Agt = \{a, b, c\}$ and the SL formula $\varphi = \langle\!\langle x\rangle\!\rangle(a, x)(b, y)\mathsf{F}\,p$ [72]. We have $\mathsf{free}(\varphi) = \{c, y\}$ because agent c is not bound to any variable after $\mathsf{F}\,p$ and variable y is not quantified. We also have $\mathsf{free}((c, z)\varphi) = \{y, z\}$ and $\mathsf{free}((a, z)\varphi) = \mathsf{free}(\varphi)$ because $c \in \mathsf{free}(\varphi)$ and $a \notin \mathsf{free}(\varphi)$. Hence, $(c, z)\varphi$ is agent-closed while $(a, z)\varphi$ is not.

Before defining the basic concepts necessary for SL semantics, we briefly explain an important attribute of SL formulas called the *alternation number*, which we need in order to be able to describe SL model checking complexity (see Table 2.1). It refers to the maximum number of quantifier switches $\langle\!\langle\cdot\rangle\!\rangle[\![\cdot]\!]$, $[\![\cdot]\!]\langle\!\langle\cdot\rangle\!\rangle$, $\langle\!\langle\cdot\rangle\!\rangle\neg\langle\!\langle\cdot\rangle\!\rangle$ or $[\![\cdot]\!]\neg[\![\cdot]\!]$ that bind a variable in a subformula that is not a sentence. To determine the alternation number of an SL formula, we first replace all its subsentences with atoms and then count the number of quantifier switches. For example, consider an interpreted system with agents $Agt = \{a, b\}$ and the SL sentence $\varphi \triangleq [\![x]\!]\langle\!\langle y\rangle\!\rangle(a, x)(b, y)\mathsf{F}\,\varphi'$ with $\varphi' \triangleq [\![x]\!]\langle\!\langle y\rangle\!\rangle(a, x)(b, y)\mathsf{X}\,p$ [72]. The alternation number of φ is 1 because φ' is a sentence and there is one quantifier switch in $[\![x]\!]\langle\!\langle y\rangle\!\rangle(a, x)(b, y)\mathsf{F}\,p_{\varphi'}$ (the subsentence φ' is replaced with an atom $p_{\varphi'}$). However, if we replaced φ' with $\varphi'' \triangleq [\![x]\!](a, x)\mathsf{X}\,p$ in φ, the alternation number of φ would be 2 because φ'' is not a sentence. Please refer to [72] for more details about this concept.

We shall now define several basic concepts relevant to SL semantics. Since we allow strategies to be *shared* among agents, we first need to slightly modify strategies (see Definition 2.16):

Definition 2.21 (Shared Strategies)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system and $A \subseteq Agt$ a non-empty set of agents. Then a partial function $f_A : Trk \rightharpoonup Act_A$, such that for all agents $i \in A$ and tracks $\tau \in \mathrm{dom}(f_A)$, $f_A(\tau) \in P_i(l_{i\mathrm{E}}(\mathsf{last}(\tau)))$, is a *shared strategy* for agents A which maps tracks to actions. $SStr_A \subseteq Trk \rightharpoonup Act_A$ denotes the set of all shared strategies for agents A. The set of all shared strategies is defined as $SStr \triangleq \bigcup_{A \subseteq Agt}^{A \neq \emptyset} SStr_A$.

Observe that an individual strategy $f_i \in Str_i$ for an agent $i \in Agt$ (Definition 2.16) is also a shared strategy (Definition 2.21), i.e. $f_i \in SStr_{\{i\}}$. So shared strategies are a generalisation of individual strategies, i.e. $Str \subseteq SStr$. We shall see that many SL concepts are generalisations of those of ATL.

In order to determine the set of strategies over which a variable can quantify, we need to know which agents are bound to it. We thus introduce the set of *sharing agents*. Intuitively, it refers to the set of agents that share a variable within a formula.

Definition 2.22 (Sharing Agents)**.** Let $\mathcal{I}$ be an interpreted system. The function $\mathsf{sharing} : SL \times Var \to 2^{Agt}$ returns the set of agents *sharing* a variable in an SL formula. It is defined inductively as follows:

1. $\mathsf{sharing}(\top, x) \triangleq \emptyset$;
2. $\mathsf{sharing}(p, x) \triangleq \emptyset$, where $p \in AP$;
3. $\mathsf{sharing}(\neg\varphi, x) \triangleq \mathsf{sharing}(\varphi, x)$;
4. $\mathsf{sharing}(\varphi_1 \,\mathsf{Op}\, \varphi_2, x) \triangleq \mathsf{sharing}(\varphi_1, x) \cup \mathsf{sharing}(\varphi_2, x)$, where $\mathsf{Op} \in \{\wedge, \vee\}$;
5. $\mathsf{sharing}(\mathsf{Op}\, \varphi, x) \triangleq \mathsf{sharing}(\varphi, x)$, where $\mathsf{Op} \in \{\mathsf{X}, \mathsf{F}, \mathsf{G}\}$;
6. $\mathsf{sharing}(\varphi_1 \,\mathsf{U}\, \varphi_2, x) \triangleq \mathsf{sharing}(\varphi_1, x) \cup \mathsf{sharing}(\varphi_2, x)$;
7. $\mathsf{sharing}(\mathsf{Qn}\, \varphi, x) \triangleq \mathsf{sharing}(\varphi, x)$, where $\mathsf{Qn} \in \{[\![y]\!], \langle\!\langle y \rangle\!\rangle : y \in Var\}$;
8. $\mathsf{sharing}((i, x)\varphi, x) \triangleq \{i\} \cup \mathsf{sharing}(\varphi, x)$;
9. $\mathsf{sharing}((i, y)\varphi, x) \triangleq \mathsf{sharing}(\varphi, x)$, where $y \in Var \setminus \{x\}$;

Note that for conciseness we assume throughout this report that every variable is quantified at most once in a given formula. This can be easily ensured by renaming variables which are not free in the formula (e.g. $(a, x)\langle\!\langle x \rangle\!\rangle(b, x)\mathsf{X}\, p \equiv (a, x)\langle\!\langle y \rangle\!\rangle(a, y)\mathsf{X}\, p \not\equiv (a, y)\langle\!\langle x \rangle\!\rangle(a, x)\mathsf{X}\, p$).

When we define the modelling relation for SL, we will somehow need to keep track of the strategies assigned to variables and agents. This information is encapsulated in an *assignment*.

Definition 2.23 (Assignments)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system. An *assignment* is a partial function $\chi : Var \cup Agt \rightharpoonup SStr$ which maps variables and agents in its domain to shared strategies. The set $Asg \subseteq Var \cup Agt \rightharpoonup SStr$ contains all possible assignments.

An assignment $\chi \in Asg$ is *complete* iff it is defined on all agents, i.e. $Agt \subseteq \mathrm{dom}(\chi)$. The set $CAsg \subseteq Asg$ contains all complete assignments.

Observe that an agent assignment $\alpha \in AAsg_A$ for a set of agents $A \subseteq Agt$ (Definition 2.16) is also an assignment (Definition 2.23), i.e. $\alpha \subseteq Asg$. Hence, SL assignments are again a generalisation of ATL agent assignments, i.e. $AAsg \subseteq Asg$. In addition, we also have $AAsg_{Agt} \subseteq CAsg$ because $\mathrm{dom}(\alpha) = Agt$ for all $\alpha \in AAsg_{Agt}$.

Assume that the history of the system has been $\rho \in Trk$. Given some strategy $f \in SStr$, we would like to determine the equivalent strategy $(f)_\rho \in SStr$ for the new history after ρ. Informally, we want to cut off the past. This is achieved by *translating* the strategy.

Definition 2.24 (Strategy Translation)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $f \in SStr_A$ a shared strategy for a set of agents $A \subseteq Agt$, and $\rho \in \mathrm{dom}(f)$ a track in its domain. Then $(f)_\rho \in SStr_A$ denotes the *translation* of f along ρ with $\mathrm{dom}((f)_\rho) \triangleq \left\{\rho' \in \mathsf{track}(\mathsf{last}(\rho)) \;\middle|\; \rho \cdot \rho'_{\geq 1} \in \mathrm{dom}(f)\right\}$ such that $(f)_\rho(\rho') \triangleq f\left(\rho \cdot \rho'_{\geq 1}\right)$ for all $\rho' \in \mathrm{dom}((f)_\rho)$.

For example, let $f \in SStr_A$ be a shared strategy for agents $A \subseteq Agt$ and $\rho = [g_0, g_1, g_2]$ and $\rho' = [g_2, g_3, g_4]$ two tracks such that $\rho \cdot \rho'_{\geq 1} = [g_0, g_1, g_2, g_3, g_4] \in \mathrm{dom}(f)$. Then we have $f([g_0, g_1, g_2, g_3, g_4]) = (f)_\rho([g_2, g_3, g_4])$. Effectively, we have cut off the past history $[g_0, g_1]$ and treat g_2 as the starting state. This notion can be easily extended to assignment translation.

Definition 2.25 (Assignment Translation). Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $\chi \in Asg$ an assignment, and $\rho \in Trk$ a track. Then $(\chi)_\rho \in Asg$ denotes the *translation* of χ along ρ such that $\mathrm{dom}((\chi)_\rho) \triangleq \mathrm{dom}(\chi)$ and $(\chi)_\rho(l) \triangleq (\chi(l))_\rho$ for all $l \in \mathrm{dom}(\chi)$.

We can now define the notion of a *play*. Intuitively, a play is the unique evolution of an interpreted system determined by the current global state and agents' strategies (encapsulated in a complete assignment).

Definition 2.26 (Plays). Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system. A path $\pi \in \mathsf{path}(g)$ starting at a global state $g \in G$ is a *play* w.r.t. a complete assignment $\chi \in CAsg$ ((χ, g)-*play*, for short) if, for all $i \geq 0$, it holds that $\pi(i+1) = t(\pi(i), \langle \chi(a)(\pi_{\leq i}) : a \in Agt \rangle)$. The function $\mathsf{play} : CAsg \times G \to Pth$ returns the (χ, g)-play $\mathsf{play}(\chi, g) \in Pth(g)$, for all pairs (χ, g) in its domain.

Once again, SL plays are very similar to ATL outcomes (see Definition 2.17). The difference between the two concepts is that $\mathsf{play}(\chi, g)$ is based on a complete assignment and therefore is unique (since the complete assignment determines the next action of every agent). In contrast, $\mathsf{out}(g, \alpha)$ is a set of paths. The two concepts are essentially equivalent if we require that $\alpha \in AAsg_{Agt}$ since $AAsg_{Agt} \subseteq CAsg$.

Informally, given a complete assignment, which contains the strategies of all agents, and the current global state, we know exactly how the whole system will evolve. This idea is formalised in the following definition.

Definition 2.27 (Global Translation). Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $\chi \in CAsg$ a complete assignment, and $g \in G$ a global state. Then the *i-th global translation* of (χ, g), with $i \geq 0$ is defined as $(\chi, g)^i \triangleq ((\chi)_{\pi_{\leq i}}, \pi(i))$, where $\pi \triangleq \mathsf{play}(\chi, g)$.

We need to define one more concept before we give the SL semantics. When an agent binding (i, x) is encountered, we must update the assignment so that the strategy assigned to agent $i \in Agt$ is the same as the one for the variable $x \in Var$. We generalise this concept of *redefinition* to an arbitrary partial function.

Definition 2.28 (Redefinition). Let A, B be two arbitrary sets, $f : A \rightharpoonup B$ a partial function, and $a \in A$, $b \in B$ elements of the sets. Then, $f[a \mapsto b] : A \rightharpoonup B$ denotes a new partial function defined on $\mathrm{dom}(f[a \mapsto b]) \triangleq \mathrm{dom}(f) \cup \{a\}$ such that $f[a \mapsto b](a) \triangleq b$ and $f[a \mapsto b](a') \triangleq f(a')$ for all $a' \in \mathrm{dom}(f) \setminus \{a\}$.

We are now finally ready to define SL semantics. Intuitively, a formula $\langle\!\langle x \rangle\!\rangle \varphi$ is true iff there exists a strategy for x such that φ is true. Conversely, a formula $[\![x]\!]\varphi$ is true iff for all strategies assigned to x we have φ. $(a, x)\varphi$ simply binds the strategy assigned to variable x to agent a.

Definition 2.29 (SL Semantics). Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an interpreted system, $g \in G$ a global state, $\varphi \in SL$ an SL formula, and $\chi \in Asg$ an assignment with $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(\chi)$. The modelling relation $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi$ is inductively defined as follows:

1. $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \top$.
2. $\mathcal{I}, \chi, g \models_{\mathrm{SL}} p$ iff $g \in h(p)$, with $p \in AP$.
3. For all formulas $\varphi, \varphi_1, \varphi_2 \in SL$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \neg\varphi$ iff $\mathcal{I}, \chi, g \not\models_{\mathrm{SL}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_1 \wedge \varphi_2$ iff $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_1$ and $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_2$;
   (c) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_1 \vee \varphi_2$ iff $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_1$ or $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_2$.

4. For all variables $x \in \mathit{Var}$ and formulas $\varphi \in \mathit{SL}$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \langle\langle x \rangle\rangle \varphi$ iff there exists a strategy $f \in \mathit{SStr}_{\mathsf{sharing}(\varphi,x)}$ such that $\mathcal{I}, \chi[x \mapsto f], g \models_{\mathrm{SL}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} [\![x]\!] \varphi$ iff for all strategies $f \in \mathit{SStr}_{\mathsf{sharing}(\varphi,x)}$ it holds that $\mathcal{I}, \chi[x \mapsto f], g \models_{\mathrm{SL}} \varphi$.
5. For all agents $i \in \mathit{Agt}$, variables $x \in \mathit{Var}$, and formulas $\varphi \in \mathit{SL}$, it holds that $\mathcal{I}, \chi, g \models_{\mathrm{SL}} (i, x)\varphi$ iff $\mathcal{I}, \chi[i \mapsto \chi(x)], g \models_{\mathrm{SL}} \varphi$.
6. For all formulas $\varphi, \varphi_1, \varphi_2 \in \mathit{SL}$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \mathsf{X}\, \varphi$ iff $\mathcal{I}, (\chi, g)^1 \models_{\mathrm{SL}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \mathsf{F}\, \varphi$ iff there is an index $i \geq 0$ such that $\mathcal{I}, (\chi, g)^i \models_{\mathrm{SL}} \varphi_2$;
   (c) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \mathsf{G}\, \varphi$ iff for all indices $i \geq 0$ it holds that $\mathcal{I}, (\chi, g)^i \models_{\mathrm{SL}} \varphi_2$;
   (d) $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there is an index $i \geq 0$ such that $\mathcal{I}, (\chi, g)^i \models_{\mathrm{SL}} \varphi_2$ and, for all indices $0 \leq j < i$ it holds that $\mathcal{I}, (\chi, g)^j \models_{\mathrm{SL}} \varphi_1$.

To our best knowledge, there are no existing model checkers for SL. The aim of this project is to develop the first such tool (see Section 1.1). The *model checking problem* for SL is formally defined as:

Given an interpreted system $\mathcal{I}$, a global state $g \in G$, an SL formula $\varphi \in \mathit{SL}$, and an assignment $\chi \in \mathit{Asg}$ with $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(\chi)$, determine whether $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi$.

In addition to model checking, we are also interested in the problem of strategy synthesis. Informally, we want to construct strategies which make a particular SL formula true. More formally, the *strategy synthesis problem* is defined as:

Given an interpreted system $\mathcal{I}$, a global state $g \in G$, and an SL formula $\varphi \in \mathit{SL}$, find an assignment $\chi \in \mathit{Asg}$ with $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(\chi)$ such that $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \varphi$.

Let us now investigate the expressive power of SL. The sample properties listed at the beginning of this section can be expressed in SL as follows:

1. $\wp_\forall\, (\mathit{open} \wedge \mathsf{X}\, \mathit{open} \wedge \mathsf{X}\,\mathsf{X}\, \mathit{open})$.
2. $\mathit{connected} \rightarrow \wp_\exists \mathsf{F}\, \neg \mathit{connected}$.
3. $\wp_\forall \mathsf{F}\, \mathit{finished}$.
4. $\wp_\exists \mathsf{F}\, \wp_\forall \mathsf{G}\, \mathit{deleted}$.
5. $\wp_\forall\, (\mathsf{G}\,\mathsf{F}\, \mathit{red} \rightarrow \mathsf{G}\,\mathsf{F}\, \mathit{green})$.
6. $[\wp_\exists \mathsf{G}\,\mathsf{F}\, \mathit{left}] \rightarrow [\wp_\exists \mathsf{G}\,\mathsf{F}\, \mathit{right}]$.
7. $\langle\langle x_{\text{firewall}} \rangle\rangle \langle\langle x_{\text{antivirus}} \rangle\rangle ([\![x_i]\!])_{i \in \mathit{Agt} \setminus \{\text{firewall}, \text{antivirus}\}} ((i, x_i))_{i \in \mathit{Agt}}\, \mathsf{G}\, \neg \mathit{hacked}$.
8. $[\![x]\!](\text{player}_1, x)(\text{player}_2, x)([\![x_i]\!](i, x_i))_{i \in \mathit{Agt} \setminus \{\text{player}_1, \text{player}_2\}}\, \mathsf{G}\, (\neg \mathit{win}_1 \wedge \neg \mathit{win}_2)$.

where $\wp_\exists \triangleq (\langle\langle x_i \rangle\rangle (i, x_i))_{i \in \mathit{Agt}}$ and $\wp_\forall \triangleq ([\![x_i]\!](i, x_i))_{i \in \mathit{Agt}}$ are two prefixes. This demonstrates that SL is more expressive than any of the previous formalisms including ATL* because it quantifies and binds strategies explicitly. However, there are other types of specifications which SL does not support. These include statements about agents' knowledge, which can be expressed using epistemic modalities.

2.2.6 Epistemic modalities

All formalisms defined so far deal only with time. However, there are other modalities (e.g. knowledge, necessity, belief, etc.) which can be expressed by extending one of the logics described in the previous subsections. In this subsection we discuss epistemic modalities expressing agents' *knowledge*.

Modelling agents' knowledge in multi-agent systems is very useful because it allows us to reason about the decision making process of individual agents as well as the interactions within a group of agents [42]. Syntactically, the knowledge of (groups of) agents is represented using epistemic connectives with the following informal meanings:

- $\mathsf{K}_i\,\varphi$ – agent $i \in Agt$ knows that φ is true (individual knowledge);
- $\mathsf{E}_A\,\varphi$ – each agent in $A \subseteq Agt$ knows that φ is true (group knowledge);
- $\mathsf{D}_A\,\varphi$ – all agents $A \subseteq Agt$ (together) know that φ is true (distributed knowledge);
- $\mathsf{C}_A\,\varphi$ – it is common knowledge[9] among agents $A \subseteq Agt$ that φ is true (common knowledge).

Any of the logics described in the previous subsections can be augmented with the epistemic connectives. For instance, if we extend Atl (see Subsection 2.2.4) with the new operators, we obtain *Alternating-Time Temporal Logic with Knowledge* (Atlk).

Definition 2.30 (Atlk Syntax)**.** Atlk *formulas* are built inductively from the set of propositional atoms AP and agents Agt, by using the following grammar, where $p \in AP$, $i \in Agt$, and $A \subseteq Agt$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\!\langle A\rangle\!\rangle \mathsf{X}\,\varphi \mid \langle\!\langle A\rangle\!\rangle \mathsf{F}\,\varphi \mid \langle\!\langle A\rangle\!\rangle \mathsf{G}\,\varphi \mid \langle\!\langle A\rangle\!\rangle [\varphi\,\mathsf{U}\,\varphi] \mid \mathsf{K}_i\,\varphi \mid \mathsf{E}_A\,\varphi \mid \mathsf{D}_A\,\varphi \mid \mathsf{C}_A\,\varphi$$

ATLK denotes the infinite set of formulas generated by the above rules.

The semantics of agents' knowledge is traditionally based on *epistemic accessibility* [61]. Intuitively, two states are *epistemically accessible* if they are indistinguishable by an agent. This is formally represented using a binary epistemic accessibility relation on states of the system. In interpreted systems (see Definition 2.5), the individual epistemic accessibility relation $\sim_i \subseteq G \times G$ of an agent $i \in Agt$ is naturally induced by its local states: two global states $g_1, g_2 \in G$ are indistinguishable by the agent iff its local states are the same, i.e. $l_{i\mathrm{E}}(g_1) = l_{i\mathrm{E}}(g_2)$. A formal definition follows.

Definition 2.31 (Epistemic Accessibilities)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i\in Agt}, I, h\right\rangle$ be an interpreted system. Let $i \in Agt$ be an agent and $A \subseteq Agt$ a set of agents. Then the *epistemic accessibility relations* on G are defined as:

- The *individual* epistemic accessibility relation $\sim_i$ of the agent i is defined by $g \sim_i g'$ if and only if $l_{i\mathrm{E}}(g) = l_{i\mathrm{E}}(g')$, i.e. the local states of agent i in global states g and g' are the same.
- The *group* epistemic accessibility relation $\sim^{\mathsf{E}}_A$ of the set of agents A is defined as $\sim^{\mathsf{E}}_A \triangleq \bigcup_{j\in A} \sim_j$, i.e. the local states of at least one agent in A in global states g and g' are the same.
- The *distributed* epistemic accessibility relation $\sim^{\mathsf{D}}_A$ of the set of agents A is defined as $\sim^{\mathsf{D}}_A \triangleq \bigcap_{j\in A} \sim_j$, i.e. the local states of all agents in A in global states g and g' are the same.
- The *common* epistemic accessibility relation $\sim^{\mathsf{C}}_A$ of the set of agents A is defined as $\sim^{\mathsf{C}}_A \triangleq \left(\sim^{\mathsf{E}}_A\right)^+$, i.e. transitive closure of group accessibility relation.

We then ascribe knowledge to agents in the following sense [42]: We say that an agent *knows* a fact iff it is true at all the worlds he considers possible, i.e. in all states that he cannot distinguish from the current one. This interpretation of knowledge is reflected in the semantics of the epistemic connectives K_i, E_A, D_A, and C_A.

Definition 2.32 (Atlk Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i\in Agt}, I, h\right\rangle$ be an interpreted system, $g \in G$ a global state, $i \in Agt$ an agent, and $A \subseteq Agt$ a set of agents. We define $\mathcal{I}, g \models_{\text{Atlk}} \varphi$ by induction on $\varphi \in$ Atlk:

- $\mathcal{I}, g \models_{\text{Atlk}} \mathsf{K}_i\,\varphi$ iff for all states $g' \in G$ such that $g \sim_i g'$ we have $\mathcal{I}, g' \models_{\text{Atlk}} \varphi$;
- $\mathcal{I}, g \models_{\text{Atlk}} \mathsf{E}_A\,\varphi$ iff for all states $g' \in G$ such that $g \sim^{\mathsf{E}}_A g'$ we have $\mathcal{I}, g' \models_{\text{Atlk}} \varphi$;
- $\mathcal{I}, g \models_{\text{Atlk}} \mathsf{D}_A\,\varphi$ iff for all states $g' \in G$ such that $g \sim^{\mathsf{D}}_A g'$ we have $\mathcal{I}, g' \models_{\text{Atlk}} \varphi$;
- $\mathcal{I}, g \models_{\text{Atlk}} \mathsf{C}_A\,\varphi$ iff for all states $g' \in G$ such that $g \sim^{\mathsf{C}}_A g'$ we have $\mathcal{I}, g' \models_{\text{Atlk}} \varphi$;
- $\mathcal{I}, g \models_{\text{Atlk}} \varphi$ is defined in the same way as $\mathcal{I}, g \models_{\text{Atl}} \varphi$ for other Atlk formulas.

[9] $\mathsf{C}_A\,\varphi = \bigwedge_{i=1}^{\infty} \mathsf{E}^i_A\,\varphi = \mathsf{E}_A\,\varphi \wedge \mathsf{E}_A\,\mathsf{E}_A\,\varphi \wedge \ldots$ i.e. each agent in A knows that φ is true, each agent in A knows that each agent in A knows that φ is true, ...

Logic	Complexity	Complexity w.r.t. $\lvert\varphi\rvert$	Complexity w.r.t. $\lvert\mathcal{I}\rvert$
Ltl	$2^{O(\lvert\varphi\rvert)}O(\lvert\mathcal{I}\rvert)$	PSpace-complete	NLogSpace-complete
Ctl	$O(\lvert\varphi\rvert \times \lvert\mathcal{I}\rvert)$	LogSpace	NLogSpace-complete
Ctl*	$2^{O(\lvert\varphi\rvert)}O(\lvert\mathcal{I}\rvert)$	PSpace-complete	NLogSpace-complete
Atl	$O(\lvert\varphi\rvert \times \lvert\mathcal{I}\rvert)$	P-complete	P-complete
Atl*	$\lvert\mathcal{I}\rvert^{2^{O(\lvert\varphi\rvert)}}$	2ExpTime-complete	P-complete
Sl	—	NonElementary	P-complete

Table 2.1: Model checking complexities of various logics [18, 38, 58, 72, 84]. $|\varphi|$ and $|\mathcal{I}|$ denote the size of the formula and the model respectively. The precise lower and upper bounds on Sl complexity with respect to the size of the formula are k-ExpSpace-hard and $(k+1)$-ExpTime respectively where k is the alternation number of φ [72].

2.2.7 Model Checking Complexity

In this subsection we discuss the computational complexity of the model checking problem. Table 2.1 shows model checking complexities for the logics described in the previous subsections. We can make the following observations:

- There is a clear correspondence between expressiveness of a logic and its model checking complexity (as expected).
- The complexity of model checking of all logics discussed in this section is P with respect to the size of the model $\mathcal{I}$.
- The complexity of model checking Ltl and Ctl* is exactly the same. Nevertheless, there are not many practical tools for Ctl* verification. This is probably due to the fact that *(i)* Ltl model checking is faster than Ctl* model checking in practice despite the theoretical complexity results, and *(ii)* branching-time logics are less natural than linear-time ones [84].
- While both Ctl and Atl formulas can be checked in time $O(|\varphi| \times |\mathcal{I}|)$, the theoretical complexity of Ctl is lower because it is related to graph reachability, whereas Atl model checking is related to AND–OR graph reachability [18].

It is important to emphasise that the size of the model $|\mathcal{I}|$ refers to the number of states (not variables). In general, adding one Boolean variable to a system doubles the size of the model. This is sometimes referred to as the *state explosion problem.* The size of a model with m Boolean variables is $O(2^m)$. More generally, the size of a model with n variables where the i-th variable has v_i possible values is $O(\prod_{i=1}^{n} v_i)$. It is often the case that they are internally represented using $m = \sum_{i=1}^{n} \lceil \log_2 v_i \rceil$ Boolean variables (e.g. when using BDDs, see Subsections 2.3.1 and 2.3.2). Since models are usually described implicitly in the form of a program (based on variables), Ctl complexity is sometimes stated as PSpace-complete with respect to program size [64].

2.2.8 Summary

In this section, we have described several well-established temporal logics as well as the relatively new Sl formalism, which is central to our project. We gave the syntax and semantics of all logics. We also briefly outlined how they can be model checked and compared their relative expressiveness on a fixed set of properties. The following diagram summarises the relationships between the logics ($L_1 \to L_2$ means that L_2 is strictly more expressive than L_1):

$$\begin{tikzcd}
 & \textsc{Atl} \arrow[dr] & & \\
\textsc{Ctl} \arrow[ur] \arrow[dr] & & \textsc{Atl}^* \arrow[r] & \textsc{Sl} \\
 & \textsc{Ctl}^* \arrow[ur] & & \\
\textsc{Ltl} \arrow[ur] & & &
\end{tikzcd}$$

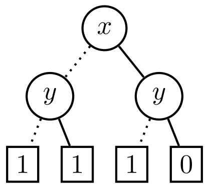
We also discussed epistemic modalities expressing agents' knowledge and provided the model checking complexities of all logics.

Verification technique	Number of states
Explicit approaches	10^{6}
Binary decision diagrams	10^{25}
Bounded model checking	10^{100}
Symmetry reduction	10^{150}
Predicate abstraction	10^{1000}

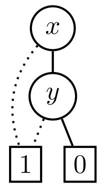
Table 2.2: Approximate state spaces that can be handled by various techniques [62].

x	y	$\neg(x \wedge y)$
0	0	1
0	1	1
1	0	1
1	1	0

(a) Truth table



(b) BDT



(c) BDD

Figure 2.1: Truth table, a binary decision tree (BDT), and a binary decision diagram (BDD) for the propositional formula $\neg(x \wedge y)$.

2.3 Verification Methods

Once we have converted a system S and a property P to a model $\mathcal{M}_S$ and a formula φ_P respectively (as described in the previous sections), we can verify whether the system indeed satisfies the property by checking [62]:

$$\mathcal{M}_S, s_0 \models \varphi_P$$

where s_0 is the initial state of the system. As we have already mentioned, it is often more convenient to calculate the set of states $\|\varphi_P\|_{\mathcal{M}_S}$ at which φ_P is true first:

$$\|\varphi_P\|_{\mathcal{M}_S} \triangleq \{s \mid \mathcal{M}_S, s \models \varphi_P\}$$

Verifying that the system satisfies the property is then equivalent to checking $s_0 \in \|\varphi_P\|_{\mathcal{M}_S}$.

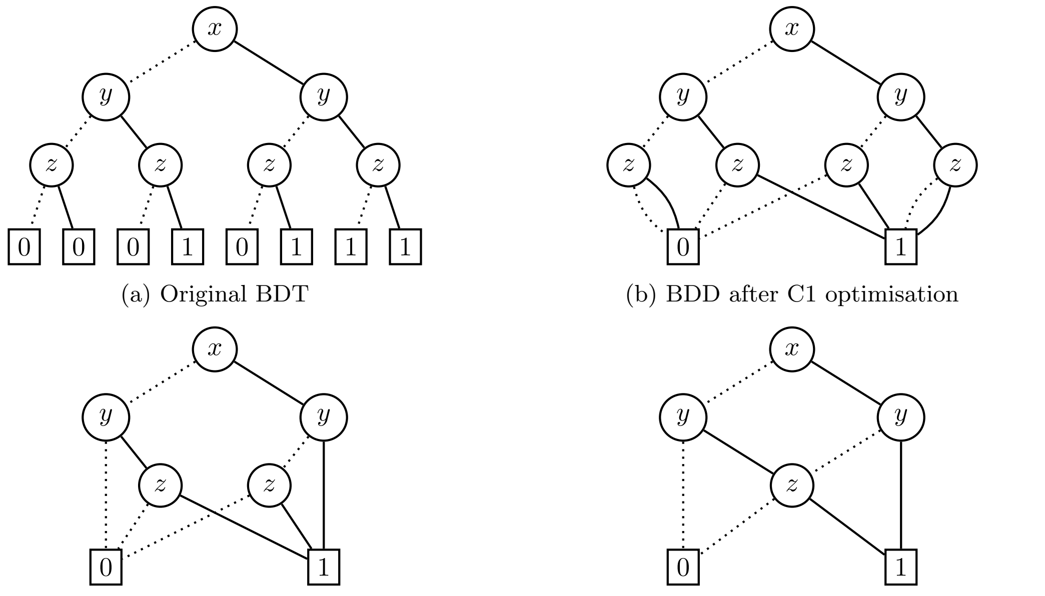
As an example demonstrating this approach, the recursive model-checking algorithm $\mathsf{SAT}_{\text{CTL}}$ for CTL was presented in the previous section (see Definition 2.12). We can use it directly to verify an arbitrary property of an arbitrary system (so-called explicit model checking). However, models are rarely given explicitly, but instead, in the form of a program. Moreover, we would like to avoid direct calculation of the set operations, which could be very slow. Therefore, other more efficient techniques have been proposed and implemented. Table 2.2 shows the maximum state spaces that can be handled by various techniques.

2.3.1 Binary Decision Diagrams

Binary decision diagrams [49] are a technique for representing propositional formulas (similarly to truth tables). Figure 2.1 shows a truth table (Figure 2.1a) and a binary decision tree (Figure 2.1b) for the propositional formula $\neg(x \wedge y)$.

Definition 2.33 (Binary Decision Trees)**.** A *binary decision tree* (BDT) is a complete binary tree whose non-terminal nodes are labelled with Boolean variables (x_1, x_2, …, x_n) and whose terminal nodes are labelled with Boolean values 0 and 1. Each non-terminal node of a BDT has two outgoing edges – one solid line and one dotted line (representing that the corresponding Boolean variable is 1 and 0 respectively). A BDT has one layer of non-terminal nodes for every Boolean variable.

A binary decision tree f represents a Boolean function $f(x_1, x_2, \dots, x_n)$. Given an assignment to all variables x_1, x_2, …, x_n occuring in f, we determine the value of the function as follows. We start at the root of f. If the current node is non-terminal with label x_i, we follow the solid line when $x_i = 1$ and the dotted line when $x_i = 0$. If the current node is terminal, the value of the function is the label of the terminal node (0 or 1).



(a) Original BDT

(b) BDD after C1 optimisation

(c) BDD after C2 optimisation (twice)

(d) Final reduced BDD after C3 optimisation

Figure 2.2: All reduction steps of the BDT for the formula $(x \wedge y) \vee (y \wedge z) \vee (z \wedge x)$ using optimisations C1–C3.

A BDT for a formula $\varphi(x_1, x_2, \ldots, x_n)$ with n variables has $2^n - 1$ non-terminal nodes and 2^n terminal nodes. Binary decision trees are thus no more efficient than truth tables (the truth table for $\varphi(x_1, x_2, \ldots, x_n)$ has 2^n entries). Fortunately, BDTs can often be compressed. Consider the BDT in Figure 2.1b. The left y node is redundant since both its edges point to terminal nodes labelled with 1 (although distinct). The dotted edge from the node labelled with x can thus point to a terminal node labelled with 1 directly instead. Intuitively, once we know that x is false, the formula $\neg(x \wedge y)$ is true regardless of the value of y. The resulting graph-based structure (Figure 2.1c) is a binary decision diagram [49, p. 364]. Binary decision diagrams (BDDs) are a generalisation of binary decision trees.

There are three different ways for reducing a BDD [49, p. 363]:

C1. Removal of duplicate terminals. We merge all terminal 0-nodes into one 1-node. Similarly, we merge all terminal 1-nodes into one 1-node.

C2. Removal of redundant tests. If both outgoing edges of a non-terminal node n point to the same non-terminal node m, we can remove node n and send all its incoming edges to node m.

C3. Removal of duplicate non-terminals. If two distinct non-terminal nodes m, n are roots of two structurally identical sub-BDDs, we can merge m and n.

The optimisations C1–C3 are demonstrated in Figure 2.2.

Definition 2.34 (Binary Decision Diagram)**.** A *binary decision diagram* (BDD) is a finite directed acyclic graph with a unique initial node whose non-terminal nodes are labelled with Boolean variables (x_1, x_2, $\ldots$, x_n) and whose terminal nodes are labelled with Boolean values 0 and 1. Each non-terminal node of a BDD has two outgoing edges – one solid line and one dotted line (representing that the corresponding Boolean variable is 1 and 0 respectively).

A BDD is *reduced* iff none of the optimisations C1–C3 can be applied to it.

BDDs are often much more space efficient than BDTs [49, p. 366]. However, they still have several drawbacks, *(i)* there may be multiple occurences of the same variable along a path (which is inefficient), and *(ii)* there is no simple way of comparing two BDDs for equivalence. Figure 2.3 shows 6 different reduced BDDs representing the same propositional formula (demonstrating the second problem). We can solve this problem by imposing an ordering on the variables [49, p. 367].

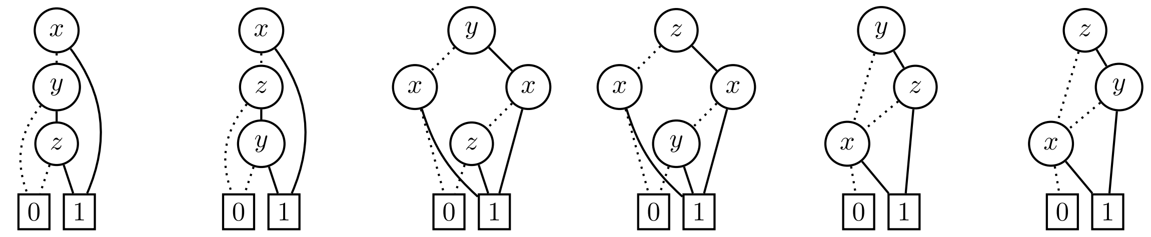

(a) $[x, y, z]$ (b) $[x, z, y]$ (c) $[y, x, z]$ (d) $[z, x, y]$ (e) $[y, z, x]$ (f) $[z, y, x]$

Figure 2.3: All possible ROBDD orderings for the propositional formula $x \lor (y \land z)$.

Definition 2.35 (Ordered Binary Decision Diagram)**.** Let $[x_1, x_2, \dots x_n]$ be an ordered list of variables without duplications and f be a BDD all of whose variables occur in the list. The BDD f has the *ordering* $[x_1, x_2, \dots, x_n]$ iff for every occurence of x_i followed by x_j along any path in f, we have $i < j$.

A BDD is an *ordered binary decision diagram* (OBDD) iff it has some ordering. An OBDD is a *reduced ordered binary decision diagram* (ROBDD) iff it is reduced.

Figure 2.3 shows ROBDD with different orderings for the same propositional formula. In general, the ordering of the chosen variable ordering has a significant impact on the size and shape of a ROBDD [49]. Note that the BDT in Figure 2.1b is ordered although it is not reduced. Although two structurally different reduced BDDs can be semantically equivalent, once we fix the ordering of the variables, the ROBDD for a given Boolean expression is unique. All OBDDs which are semantically equivalent thus have a *canonical form* – the reduced form, which can be obtained by applying optimisations C1–C3 [62].

To sum up, ROBDDs have the following advantages (compared to truth tables) [49]:

1. They provide a *compact representation* of complex Boolean formulas. Nevertheless, the size of the ROBDD representing certain Boolean functions (e.g. integer multiplication [49, p. 381]) is still exponential.

2. They have a *canonical form* allowing very simple equivalence, satisfiability, and validity checking.

3. They support efficient calculation of various *Boolean operations* (e.g. conjunction). Please refer to [49, p. 372] for more details about the corresponding algorithms.

These properties make ROBDDs suitable for use in verification as we describe in the next section.

2.3.2 Symbolic Model Checking

We will now describe how BDDs described in the previous subsection can be used for symbolic model checking. We will proceed as follows:

1. **Translate states to conjunctions of literals.** Consider m Boolean variables $x_1, \dots x_m$. They can encode 2^m different values[10]. Given a set S of states, we need n Boolean variables such that $2^n \leq |S|$ [62]. The smallest n satisfying this condition is in general $n = \lceil \log_2 |S| \rceil$. Each state is then uniquely represented by a conjunct of the variables (e.g. $f_{s_1} = (\neg x_1 \land \neg x_2 \land \dots \land \neg x_n)$). Table 2.3a shows how this can be done.

2. **Translate sets of states to Boolean formulas.** We can encode a set of states $X = \{s_1, s_2, \dots, s_m\}$ as a Boolean formula $f_X = \bigvee_{i=1}^{m} f_i = f_1 \lor f_2 \lor \dots \lor f_m$. Since each disjunct f_i is a conjunction of literals, the resulting formula f_X is in DNF[11]. Table 2.3b shows how this can be done.

3. **Translate set operations to Boolean operations.** Let $A, B \subseteq S$ be arbitrary sets of states represented by Boolean formulas f_A, f_B. We can express set operations as follows: $f_{A \cup B} = f_A \lor f_B$ (union), $f_{A \cap B} = f_A \land f_B$ (intersection), $f_{A \setminus B} = f_A \land \neg f_B$ (difference), and $f_{A'} = f_S \land \neg f_A$ (complement).

[10]We can think of these values as binary numbers from $x_1 = 0, \dots, x_m = 0$ up to $x_1 = 1, \dots, x_m = 1$.

[11]Disjunctive normal form.

State	Equivalent formula
s_1	$\neg x_1 \wedge \neg x_2 \wedge \neg x_3$
s_2	$\neg x_1 \wedge \neg x_2 \wedge x_3$
s_3	$\neg x_1 \wedge x_2 \wedge \neg x_3$
s_4	$\neg x_1 \wedge x_2 \wedge x_3$
s_5	$x_1 \wedge \neg x_2 \wedge \neg x_3$
s_6	$x_1 \wedge \neg x_2 \wedge x_3$

(a) States.

Set	Equivalent formula (simplified)
$\emptyset$	0
$\{s_1\}$	$\neg x_1 \wedge \neg x_2 \wedge \neg x_3$
$\{s_1, s_2\}$	$\neg x_1 \wedge \neg x_2$
$\{s_2, s_3\}$	$\neg x_1 \wedge ((\neg x_2 \wedge x_3) \vee (x_2 \wedge \neg x_3))$
$\{s_4, s_5, s_6\}$	$(\neg x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \neg x_2)$
S	$\neg x_1 \vee \neg x_2$

(b) Sets of states.

Table 2.3: Possible encoding of states using Boolean formulas ($S = \{s_1, s_2, s_3, s_4, s_5, s_6\}$). Formulas for sets of states are obtained by disjuntion, e.g. $f_{\{s_1,s_2\}} = f_{s_1} \vee f_{s_2} = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge \neg x_2 \wedge x_3) = \neg x_1 \wedge \neg x_2$. Note that since certain conjuncts do not represent any states (e.g. $x_1 \wedge x_2 \wedge x_3$), we do not care if they are in a set and thus we can for example encode the set of all states simply as $f_S = 1$. Also note that this is not the only possible encoding of states.

4. **Translate relations to Boolean formulas** Let $R = \{(s_{1a}, s_{1b}), (s_{2a}, s_{2b}), \ldots, (s_{ma}, s_{mb})\} \subseteq S \times S$ be a binary relation. We can encode it as a Boolean expression $f_R = \bigvee_{i=1}^{m} \left(f_{s_{ia}} \wedge f'_{s_{ib}}\right) = \left(f_{s_{1a}} \wedge f'_{s_{1b}}\right) \vee \left(f_{s_{2a}} \wedge f'_{s_{2b}}\right) \vee \cdots \vee \left(f_{s_{ma}} \wedge f'_{s_{mb}}\right)$, where f'_{s_i} is obtained from f_{s_i} by replacing variables $x_1, x_2, \ldots, x_n$ with variables $x'_1, x'_2, \ldots, x'_n$. Figure 2.4 shows how a transition relation can be encoded. Note that we need $2n$ variables to represent the binary relation. Relations of higher arities can be encoded in a similar way.

This allows us to express a model $\mathcal{M}_S$ using BDDs for sets of states, transition relations, protocols, etc. The function $\mathsf{SAT}^{\mathcal{M}_S}(\varphi_P)$ which recursively calculates the set of states of $\mathcal{M}_S$ satisfying a formula φ_P can be easily modified to perform the corresponding set operations on BDDs (instead of explicitly on sets of states).

To give a complete picture, we explain how the predecessor functions $\mathsf{pre}_\exists, \mathsf{pre}_\forall : 2^S \to 2^S$ are translated to BDD operations. Intuitively, given a set of states $X \subseteq S$, $\mathsf{pre}_\exists(X)$ is the set of states that can reach some state in X in one step and $\mathsf{pre}_\forall(X)$ is the set of states such that all states reachable in one step from them are in X. Formally, the general definitions of the functions are[12]:

$$\mathsf{pre}_\exists(X) \triangleq \{s \in S \mid \exists s' \in S.\, R(s, s') \wedge s' \in X\}$$
$$\mathsf{pre}_\forall(X) \triangleq \{s \in S \mid \forall s' \in S.\, R(s, s') \to s' \in X\}$$

We can observe that:

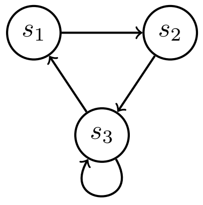
$$\begin{aligned}
\mathsf{pre}_\forall(X) &\triangleq \{s \in S \mid \forall s' \in S.\, R(s, s') \to s' \in X\} \\
&= S \setminus \{s \in S \mid \neg(\forall s' \in S.\, R(s, s') \to s' \in X)\} \\
&= S \setminus \{s \in S \mid \exists s' \in S.\, \neg(R(s, s') \to s' \in X)\} \\
&= S \setminus \{s \in S \mid \exists s' \in S.\, R(s, s') \wedge s' \notin X\} \\
&= S \setminus \{s \in S \mid \exists s' \in S.\, R(s, s') \wedge s \in S \setminus X\} \\
&= S \setminus \mathsf{pre}_\exists(S \setminus X)
\end{aligned}$$

Hence, we only need to explain how $\mathsf{pre}_\exists$ can be computed using BDDs. Let $X \subseteq S$ be an arbitrary set of states, $R \subseteq S \times S$ a transition relation, and f_X, f_R their Boolean representation (e.g. BDDs). We calculate $f_{\mathsf{pre}_\exists(X)}$ as follows[13] [62]:

1. Obtain f'_X by replacing all variables x_i with their primed versions x'_i. Intuitively, we convert X from the set of *current states* to the set of *next states.*

2. Combine the set of next states with the transition relation $f_R \wedge f'_X$. The result now already contains the set of new current states $\mathsf{pre}_\exists(X)$. However, it also still contains the next states X.

[12]We defined the predecessor functions on interpreted systems in Definition 2.11.
[13]Please refer to [49] for details on how these operations can be efficiently performed on BDDs.



(a) Model.

s	x_1	x_2
s_1	0	0
s_2	0	1
s_3	1	0
–	1	1

(b) State encoding.

s	s'	x_1	x_2	x'_1	x'_2	R
s_1	s_1	0	0	0	0	0
s_1	s_2	0	0	0	1	1
s_1	s_3	0	0	1	0	0
s_1	–	0	0	1	1	–
s_2	s_1	0	1	0	0	0
s_2	s_2	0	1	0	1	0
s_2	s_3	0	1	1	0	1
s_2	–	0	1	1	1	–
s_3	s_1	1	0	0	0	1
s_3	s_2	1	0	0	1	0
s_3	s_3	1	0	1	0	1
s_3	–	1	0	1	1	–
–	s_1	1	1	0	0	–
–	s_2	1	1	0	1	–
–	s_3	1	1	1	0	–
–	–	1	1	1	1	–

(c) Transition table.

Figure 2.4: Sample encoding of the transition relation $R = \{(s_1, s_2), (s_2, s_3), (s_3, s_1), (s_3, s_3)\}$ of a model with states $S = \{s_1, s_2, s_3\}$ [62]. The Boolean formula representing the transition is $f_R = (\neg x_1 \wedge \neg x_2 \wedge \neg x'_1 \wedge x'_2) \vee (\neg x_1 \wedge x_2 \wedge x'_1 \wedge \neg x'_2) \vee (x_1 \wedge \neg x_2 \wedge \neg x'_1 \wedge \neg x'_2) \vee (x_1 \wedge \neg x_2 \wedge x'_1 \wedge \neg x'_2)$. The value "–" (so-called "don't care") represents combinations encoding invalid pairs of states.

3. Calculate $f_{\mathsf{pre}_\exists(X)} = \exists x'_1 \exists x'_2 \dots \exists x'_n \,(f_R \wedge f'_X)$. We are effectively removing the dependence of the current states on the next states.

2.3.3 Summary

In this section, we discussed how model checking can be performed symbolically using efficient operations on BDDs. Compared to explicit approaches, which can handle up to 10^6 states, BDDs support state spaces with up to 10^{25} states (see Table 2.2). We will use BDDs in later parts of this report to describe efficient symbolic implementations of our novel model checking algorithms for fragments of SL (see Subsections 4.2.4 and 5.2.4).

2.4 Automata and Games

Model checking and strategy synthesis for a specification in a particular language (e.g. LTL) can often be reduced to solving an infinite two-player game [77]. In fact, we will use this approach to provide a novel algorithm for the verification of SL[1G] in Chapter 5. This section presents the relevant concepts.

2.4.1 ω-Automata

Automata on infinite words have played a central role in specifying and verifying reactive systems (systems which do not terminate, e.g. operating systems) since their introduction in the 1960's [43]. Informally, finite ω-automata are finite state machines which accept *infinite words* (i.e. infinite sequences of symbols) according to some acceptance condition. Firstly, we formally define the concept of finite and infinite words [43].

Definition 2.36 (Words)**.** Let Σ be a finite alphabet with symbols $a, b, c, \dots \in \Sigma$. Then,

- Σ^* is the set of all *finite words* over Σ. We use small letters $u, v, w, \dots \in \Sigma^*$ to denote finite words and write $u = u(0)u(1)\dots u(n)$ with $u(i) \in \Sigma$ and length $|u| = n$.
- Σ^ω is the set of all *infinite words* over Σ. We use small greek letters $\alpha, \beta, \gamma, \dots \in \Sigma^\omega$ to denote infinite words and write $\alpha = \alpha(0)\alpha(1)\dots$ with $\alpha(i) \in \Sigma$ and length $|\alpha| = \omega$.

Given an ω-word $\alpha \in \Sigma^\omega$, the set of symbols occurring in α is:

$$\mathrm{Occ}(\alpha) = \{a \in \Sigma \mid \exists i.\alpha(i) = a\}$$

and the set of symbols occurring infinitely often in α is:

$$\mathrm{Inf}(\alpha) = \{a \in \Sigma \mid \forall i \exists j > i.\alpha(j) = a\}$$

An automaton (both finite and infinite) can be decomposed into a semi-automaton and an acceptance condition [77]. We will introduce the two concepts separately. Intuitively, a semi-automaton is a finite state machine whose edges are labelled with the symbols of a finite alphabet.

Definition 2.37 (Semi-automaton)**.** Let S be a finite set of states and Σ an alphabet. Then $\mathfrak{A} = (S, \Sigma, I, R)$ is a *semi-automaton* where $I \subseteq S$ is a set of initial states and $R \subseteq S \times \Sigma \times S$ a transition relation.

A semi-automaton $\mathfrak{A}$ is *deterministic* iff *(i)* $I = \{s_I\}$ (there is a unique initial state) and *(ii)* for all $s \in S, a \in \Sigma$ we have $|\{s' \in S \mid R(s, a, s')\}| = 1$ (there is exactly one next state for every state and input). Otherwise, $\mathfrak{A}$ is *non-deterministic.*

The transitions of a semi-automaton can be (equivalently) defined as a transition function $\delta : S \times \Sigma \to 2^S$. In the case of a *deterministic* semi-automaton, the transition function is simply $\delta : S \times \Sigma \to S$. Therefore, we will sometimes write a deterministic semi-automaton as $\mathfrak{A} = (S, \Sigma, s_I, \delta)$.

The acceptance component of an automaton is defined with respect to the set of runs of the underlying semi-automaton on an infinite word.

Definition 2.38 (Run)**.** Let $\mathfrak{A} = (S, \Sigma, I, R)$ be a semi-automaton and $\alpha \in \Sigma^\omega$ an infinite word. A *run* of $\mathfrak{A}$ on α is an infinite word $\beta \in S^\omega$ such that *(i)* $\beta(0) \in I$ and *(ii)* for all $i \geq 0$ we have $R(\beta(i), \alpha(i), \beta(i+1))$.

There are various acceptance conditions with different expressive power and translation complexity[14]. Here we present only the most common ones [43].

Definition 2.39 (Acceptance)**.** Let $\mathfrak{A} = (S, \Sigma, I, R)$ be a semi-automaton. An infinite word $\alpha \in \Sigma^\omega$ is accepted (according to the relevant acceptance condition) iff there exists a run $\beta \in S^\omega$ of $\mathfrak{A}$ on α such that:

- **Büchi acceptance condition** $F \subseteq S$ (a set of states):
$$\mathrm{Inf}(\beta) \cap F \neq \emptyset$$
- **Generalised Büchi acceptance condition** $F_1, \ldots, F_n \subseteq S$ (sets of states):
$$\forall i \in \{1, \ldots, k\}.\, \mathrm{Inf}(\beta) \cap F_i \neq \emptyset$$
- **Muller acceptance condition** $\mathcal{F} \subseteq 2^S$ (a set of state sets):
$$\mathrm{Inf}(\beta) \in \mathcal{F}$$
- **Rabin acceptance condition** $\Omega = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ where $E_i, F_i \subseteq S$ (a family of pairs of state sets):
$$\exists i \in \{1, \ldots, k\}.\ (\mathrm{Inf}(\beta) \cap E_i = \emptyset) \wedge (\mathrm{Inf}(\beta) \cap F_i \neq \emptyset)$$
- **Streett acceptance condition** $\Omega = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ where $E_i, F_i \subseteq S$ (a family of pairs of state sets):
$$\forall i \in \{1, \ldots, k\}.\ (\mathrm{Inf}(\beta) \cap E_i \neq \emptyset) \vee (\mathrm{Inf}(\beta) \cap F_i = \emptyset)$$

[14]All automata in Definition 2.39 except for deterministic (generalised) Büchi automata have the same expressive power. However, translations between the classes often incur an exponential blow-up. Please refer to [43, Section 1.7] and [83] for more details.

- **Parity acceptance condition** $c : S \to \{0, \dots, k\}$ with $k \in \mathbb{N}$ (a colouring function):

$$\min \{c(s) \mid s \in \mathrm{Inf}(\beta)\} \text{ is even}$$

- **Generalised parity acceptance condition** $c_1, \dots, c_n : V \to \{0, \dots, k_i\}$ with $k_i \in \mathbb{N}$ (a family of colouring functions):

$$\forall i \in \{1, \dots, n\}. \min \{c_i(s) \mid s \in \mathrm{Inf}(\beta)\} \text{ is even}$$

Note that the acceptance condition F of Büchi automata is often referred to as a *fairness condition.* Finally, we are ready to define an ω-automaton as a combination of a semi-automaton and an acceptance condition:

Definition 2.40 (ω-Automata)**.** Let (S, Σ, I, R) be a semi-automaton and $\mathcal{F}$ an acceptance condition. Then $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ is an (existential[15]) *ω-automaton.* $\mathsf{Lang}(\mathfrak{A}) \triangleq \{\alpha \in \Sigma^\omega \mid \mathfrak{A} \text{ accepts } \alpha\}$ is the *language* of $\mathfrak{A}$.

Clearly, Büchi automata can be transformed trivially to equivalent generalised Büchi automata by setting $k = 1$ and $F_1 = F$. We also show how a generalised Büchi automaton can be converted to an equivalent Büchi automaton. Let $\mathfrak{A} = (S, \Sigma, I, R, \langle F_1, \dots, F_k \rangle)$ be a generalised Büchi automaton [77,83]. We construct a deterministic Büchi automaton $\mathfrak{D} = (\{q_1, \dots, q_k, q_a\}, S, q_1, \delta, \{q_a\})$ with the transition function δ defined as:

$$\delta(q_a, s) \triangleq q_1$$
$$\delta(q_i, s) \triangleq \begin{cases} q_a & \text{if } s \in F_i \wedge i = k \\ q_{i+1} & \text{if } s \in F_i \wedge i < k \\ q_i & \text{if } s \notin F_i \end{cases}$$

The automaton $\mathfrak{D}$ has $k + 1$ states and is shown in Figure 2.5. Whenever it is in state q_i, it "waits" until some state in F_i is encountered. Hence, if q_a is visited infinitely often, some state in each of the sets $F_1, \dots, F_k$ must be visited infinitely often. Conversely, if some state in each of the sets $F_1, \dots F_k$ is visited infinitely often, the state q_a will be visited infinitely often.

The Büchi automaton $\mathfrak{B}$ equivalent to $\mathfrak{A}$ is obtained using the *automaton product* of $\mathfrak{A}$ (without the generalised Büchi acceptance condition) and $\mathfrak{D}$:

$$\mathfrak{B} = \mathfrak{A} \times \mathfrak{D} = \langle S \times \{q_1, \dots, q_n, q_a\}, \Sigma, I \times \{q_1\}, R \times \delta, S \times \{q_a\} \rangle$$

Note that since $\mathfrak{D}$ is deterministic, the resulting Büchi automaton $\mathfrak{B}$ is deterministic if and only if $\mathfrak{A}$ is deterministic. Some other properties (e.g. unambiguity [77], see Definition 2.44) are also preserved.

2.4.2 Symbolically Represented ω-Automata

ω-automata can soon become too large to store explicitly. Therefore, it is often useful to represent them symbolically using BDDs (see Subsection 2.3.1). In this subsection, we explain how to do that.

Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be an ω-automaton. We proceed as follows:

1. We represent the sets S and Σ using finite disjoint sets of variables V_S and V_Σ respectively. Furthermore, we introduce another set of variables $V'_S = \{v' \mid v \in V_S\}$ so that we could represent the binary transition relation. Subsection 2.3.2 explains how to do this in more detail.

2. We encode the initial set of states $I \subseteq S$ as a propositional formula over V_S.

3. We encode the transition relation $R \subseteq S \times \Sigma \times S$ as a propositional formula over $V_S \cup V_\Sigma \cup V'_S$.

[15] All ω-automata in this thesis are *existential* because we require the acceptance condition to be satisfied by *some* run (see Definition 2.39). Some authors consider $\forall$-automata instead, which accept an infinite word if *all* runs of the automaton on the word are accepting [67].

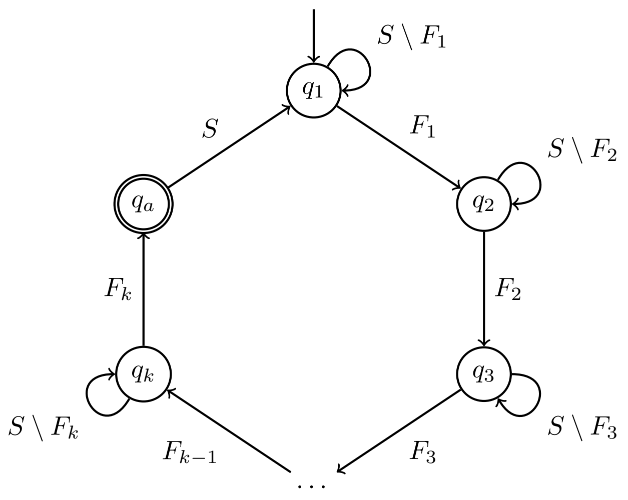


Figure 2.5: The deterministic Büchi automaton $\mathfrak{D} = (\{q_1, \dots, q_k, q_a\}, S, \delta, q_1, \{q_a\})$ for converting a generalised Büchi automaton $\mathfrak{A} = (S, \Sigma, R, I, \langle F_1, \dots, F_k\rangle)$ to a Büchi automaton. Note that accepting states have double borders and edges are labelled with sets of states (rather than individual states).

4. We encode the acceptance condition $\mathcal{F}$ as a propositional formula (or a set thereof depending on the class of the automaton) over V_S.

The following definition summarises the symbolic representation of ω-automata:

Definition 2.41 (Automaton Formula)**.** Let V_S and V_Σ be two finite disjoint sets of Boolean variables, $\mathcal{I}$ a propositional formula over V_S, $\mathcal{R}$ a propositional formula over $V_S \cup V_\Sigma \cup \{v' \mid v \in V_S\}$, and $\mathcal{F}$ a propositional formula (or a set thereof) over V_S. Then $\mathcal{A}_\exists(V_S, V_\Sigma, \mathcal{I}, \mathcal{R}, \mathcal{F})$ in an (existential) automaton formula.

We often need to combine automata by means of automaton product. This can be done easily when they are represented symbolically. Let $\mathfrak{A}_1 = \mathcal{A}_\exists(V_{S_1}, V_{\Sigma_1}, \mathcal{I}_1, \mathcal{R}_1, \mathcal{F}_1)$ and $\mathfrak{A}_2 = \mathcal{A}_\exists(V_{S_2}, V_{\Sigma_2}, \mathcal{I}_2, \mathcal{R}_2, \mathcal{F}_2)$ be two automaton formulas. The automaton formula representing their product is[16]:

$$\mathfrak{A}_1 \times \mathfrak{A}_2 = \mathcal{A}_\exists(V_{S_1} \cup V_{S_2}, V_{\Sigma_1} \cup V_{\Sigma_2}, \mathcal{I}_1 \wedge \mathcal{I}_2, \mathcal{R}_1 \wedge \mathcal{R}_2, \mathcal{F}_1 \cup \mathcal{F}_2)$$

2.4.3 Translating Ltl Formulas to ω-automata

There is clearly a resemblance between Ltl formulas and ω-automata. Both identify infinite paths which are accepted according to some criterion. Therefore, it should not come as a surprise that arbitrary Ltl formulas can be translated to finite ω-automata which are equivalent in the sense that they accept precisely those infinite paths which satisfy the corresponding formulas [86]. In fact, as we have already mentioned in Subsection 2.2.1, many modern model checkers including NuSMV (discussed in Subsection 2.5.4) first translate an Ltl formula to a generalised Büchi automaton (using the so-called tableau construction [34]) and then perform Ctl model checking with fairness constraints.

In this subsection we describe the *standard translation* [83] of Ltl formulas to non-deterministic generalised Büchi automata. As explained in Subsection 2.4.1, it is straightforward to translate a generalised Büchi automaton to a Büchi automaton. Firstly, we define what it means for an Ltl formula and an ω-automaton to be equivalent.

Definition 2.42 (Ltl-ω-Automaton Equivalence)**.** Fix a finite non-empty set of atomic propositions AP. Let $\varphi \in \mathit{LTL}$ be an Ltl formula and $\mathfrak{A} = \left(S, 2^{AP}, I, R, \mathcal{F}\right)$ an ω-automaton. We say that φ and $\mathfrak{A}$ are *equivalent* if for every interpreted system $\mathcal{I}$ and infinite path $\pi \in \mathit{Pth}$ in $\mathcal{I}$, we have:

$$\mathcal{I}, \pi \models_{\text{LTL}} \varphi \quad \text{iff} \quad \mathfrak{A} \text{ accepts } h^{-1}(\pi)$$

where $h^{-1}(\pi) = h^{-1}(\pi(0))h^{-1}(\pi(1))\dots$ and $h^{-1}(g) = \{p \in AP \mid g \in h(p)\}$ for $g \in G$.

[16]We assume that $\mathfrak{A}_1$ and $\mathfrak{A}_2$ are generalised Büchi automata. Otherwise, the combined acceptance condition might be different from $\mathcal{F}_1 \cup \mathcal{F}_2$.

We are now ready to describe how LTL formulas are translated to equivalent ω-automata. Figure 2.6 shows a recursive bottom-up algorithm (referred to as *standard translation*) which takes as input an LTL formula ϕ and returns a non-deterministic generalised Büchi automaton formula $\mathcal{A}_\exists(Q_\phi, AP, \mathcal{I}_\phi, \mathcal{R}_\phi, \mathcal{F}_\phi)$ equivalent to ϕ.

Note that the standard translation can violate the constraints on the formulas $\mathcal{I}$ and $\mathcal{F}$ in Definition 2.41, which require that they range over the state variables V_S only. It turns out that the constraint on $\mathcal{I}$ can be temporarily relaxed during the construction as it can be satisfied again at any time by simply adding an extra state variable q:

$$\mathcal{A}_\exists(V_S \cup \{q\}, V_\Sigma, q, \mathcal{R} \wedge (q \leftrightarrow \mathcal{I}), \mathcal{F})$$

This is necessary only *(i)* when a temporal operator is encountered[17] and *(ii)* at the very end so that the final automaton formula satisfies the constraints. Thanks to temporarily relaxing the constraints, we get the result summarised in the following proposition[18].

Proposition 2.2. Let $\varphi \in LTL$ be an arbitrary LTL formula with t temporal operators. We can construct a symbolic representation of a non-deterministic generalised Büchi automaton $\mathcal{A}_\exists(Q, AP, \mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi)$ equivalent to φ in time $O(|\varphi|)$ with $|Q| \leq 2t + 1$ state variables. The automaton will have at most 2^{2t+1} states.

In order to give some intuition into how the standard translation works, we will now discuss two examples. Firstly, we consider the LTL formula $\mathsf{X}\,p$. The automaton formula equivalent to p is $\mathfrak{A}_p = \mathcal{A}_\exists(\emptyset, AP, p, \top, \emptyset)$. Since we will apply the temporal operator X, we have to ensure that the initial state formula contains only state variables. Therefore, we add an extra state variable q_1 and obtain the automaton formula (also equivalent to p) $\mathfrak{A}'_p = \mathcal{A}_\exists(\{q_1\}, AP, q_1, q_1 \leftrightarrow p, \emptyset)$. Finally, we apply the rule for X and obtain the automaton formula $\mathfrak{A}_{\mathsf{X}\,p} = \mathcal{A}_\exists(\{q_1, q_2\}, AP, q_2, (q_1 \leftrightarrow p) \wedge (q_2 \leftrightarrow q'_1), \emptyset)$. The automaton is shown in Figure 2.7a. Intuitively, the variables q_1 and q_2 stand for the LTL formulas p and $\mathsf{X}\,p$ respectively. This is expressed in the transition relation by the fact that q_1 is true whenever p is true and q_2 is true whenever q_1 will be true in the next state.

Secondly, we consider the LTL formula $a \,\mathsf{U}\, b$. The automaton formulas equivalent to the atoms a and b are $\mathfrak{A}_a = \mathcal{A}_\exists(\emptyset, AP, a, \top, \emptyset)$ and $\mathfrak{A}_b = \mathcal{A}_\exists(\emptyset, AP, b, \top, \emptyset)$ respectively. The initial state formula of the automaton equivalent to the right subformula of $a \,\mathsf{U}\, b$ must contain only state variables (so that the associated fairness formula would contain only state variables). Therefore, we introduce an extra state variable q_1 and obtain the automaton formula $\mathfrak{A}'_b = \mathcal{A}_\exists(\{q_1\}, AP, q_1, q_1 \leftrightarrow b, \emptyset)$. Before applying the operator U, we calculate the automaton product $\mathfrak{A}_a \times \mathfrak{A}'_b = \mathcal{A}_\exists(\{q_1\}, AP, a \wedge q_1, q_1 \leftrightarrow b, \emptyset)$. We now introduce another extra variable q_2 which should be true whenever $a \,\mathsf{U}\, b$ is true. Intuitively, q_2 should be true when b is true (equivalent to q_1) or a is true and q_2 is true in the next state. Hence, we get the following recursive relation[19]:

$$q_2 \leftrightarrow q_1 \vee a \wedge q'_2$$

However, this specification is not sufficient because it is also satisfied by a path where a holds forever from the current state onwards. In other words, the specification admits all paths satisfying the LTL formula $a \,\mathsf{W}\, b \equiv (a \,\mathsf{U}\, b) \vee \mathsf{G}\, a$. In order to enforce that b (q_1) is eventually true (in case q_2 is true in the current state), we must add the fairness constraint:

$$q_2 \rightarrow q_1$$

We show that given an arbitrary infinite path along which the fairness condition is satisfied infinitely often, q_2 is true in the first state of the path if and only if $a \,\mathsf{U}\, b$ holds on the path:

$\Rightarrow$: Assume that q_2 is true in the first state and the fairness constraint is satisfied infinitely often. Then, either[20] q_2 is infinitely often false (in which case b must be true after a finite number of steps

[17]Note that when calculating the automaton formula for $\varphi \,\mathsf{U}\, \psi$ (in Figure 2.6), $\mathcal{I}_\varphi$ need not contain only state variables because it is neither used in a fairness constraint (like $\mathcal{I}_\psi$ for U and $\mathcal{I}_\varphi$ for F, G) nor are its variables switched between the current and the next state (like $\mathcal{I}_\varphi$ for X). Hence, every temporal operator will incur at most two new state variables in total.

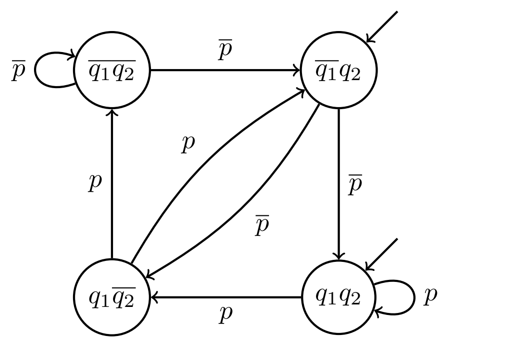
[18]The upper bound for the number of state variables (and consequently states) is slightly tighter than in Proposition 1 of [77].

[19]Note that this relation is very similar to the fixpoint rule for $\mathsf{SAT}^{\mathcal{I}}_{\mathrm{CTL}}(\mathsf{E}[\varphi_1 \,\mathsf{U}\, \varphi_2])$ in Definition 2.12.

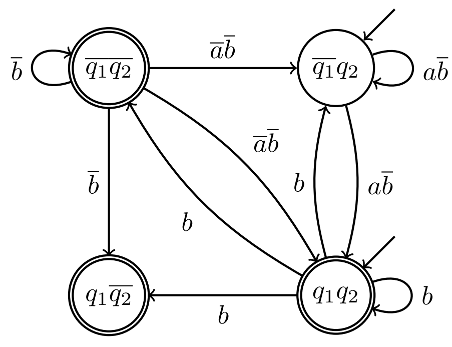
[20]We use the equivalence $q_2 \rightarrow q_1 \equiv \neg q_2 \vee q_1$ and the fact that a disjunction fairness constraint $\mathsf{G}\,\mathsf{F}\,(\varphi \vee \psi)$ is equivalent to the disjunction of fairness constraints $(\mathsf{G}\,\mathsf{F}\,\varphi) \vee (\mathsf{G}\,\mathsf{F}\,\psi)$.

```
function GenBüchi($\Phi$)
    switch $\Phi$ do
        case $p \in AP$:
            return $\mathcal{A}_\exists(\emptyset, AP, p, \top, \emptyset)$
        case $\neg\varphi$:
            $\mathcal{A}_\exists(Q_\varphi, AP, \mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi) \equiv$ GenBüchi($\varphi$)
            return $\mathcal{A}_\exists(Q_\varphi, AP, \neg\mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi)$
        case $\varphi \wedge \psi$:
            $\mathcal{A}_\exists(Q_\Phi, AP, \mathcal{I}_\varphi \wedge \mathcal{I}_\psi, \mathcal{R}_\Phi, \mathcal{F}_\Phi) \equiv$ GenBüchi($\varphi$) $\times$ GenBüchi($\psi$)
            return $\mathcal{A}_\exists(Q_\Phi, AP, \mathcal{I}_\varphi \wedge \mathcal{I}_\psi, \mathcal{R}_\Phi, \mathcal{F}_\Phi)$
        case $\varphi \vee \psi$:
            $\mathcal{A}_\exists(Q_\Phi, AP, \mathcal{I}_\varphi \wedge \mathcal{I}_\psi, \mathcal{R}_\Phi, \mathcal{F}_\Phi) \equiv$ GenBüchi($\varphi$) $\times$ GenBüchi($\psi$)
            return $\mathcal{A}_\exists(Q_\Phi, AP, \mathcal{I}_\varphi \vee \mathcal{I}_\psi, \mathcal{R}_\Phi, \mathcal{F}_\Phi)$
        case $\mathsf{X}\,\varphi$:
            $\mathcal{A}_\exists(Q_\varphi, AP, \mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi) \equiv$ GenBüchi($\varphi$)
            $q :=$ NewVar
            return $\mathcal{A}_\exists(Q_\varphi \cup \{q\}, AP, q, \mathcal{R}_\varphi \wedge (q \leftrightarrow \mathsf{X}\mathcal{I}_\varphi), \mathcal{F}_\varphi)$
        case $\mathsf{F}\,\varphi$:
            $\mathcal{A}_\exists(Q_\varphi, AP, \mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi) \equiv$ GenBüchi($\varphi$)
            $q :=$ NewVar
            return $\mathcal{A}_\exists(Q_\varphi \cup \{q\}, AP, q, \mathcal{R}_\varphi \wedge (q \leftrightarrow \mathcal{I}_\varphi \vee \mathsf{X}\,q), \mathcal{F}_\varphi \cup \{q \rightarrow \mathcal{I}_\varphi\})$
        case $\mathsf{G}\,\varphi$:
            $\mathcal{A}_\exists(Q_\varphi, AP, \mathcal{I}_\varphi, \mathcal{R}_\varphi, \mathcal{F}_\varphi) \equiv$ GenBüchi($\varphi$)
            $q :=$ NewVar
            return $\mathcal{A}_\exists(Q_\varphi \cup \{q\}, AP, q, \mathcal{R}_\varphi \wedge (q \leftrightarrow \mathcal{I}_\varphi \wedge \mathsf{X}\,q), \mathcal{F}_\varphi \cup \{\mathcal{I}_\varphi \rightarrow q\})$
        case $\varphi\,\mathsf{U}\,\psi$:
            $\mathcal{A}_\exists(Q_\Phi, AP, \mathcal{I}_\varphi \wedge \mathcal{I}_\psi, \mathcal{R}_\Phi, \mathcal{F}_\Phi) \equiv$ GenBüchi($\varphi$) $\times$ GenBüchi($\psi$)
            $q :=$ NewVar
            return $\mathcal{A}_\exists(Q_\Phi \cup \{q\}, AP, q, \mathcal{R}_\Phi \wedge (q \leftrightarrow \mathcal{I}_\psi \vee \mathcal{I}_\varphi \wedge \mathsf{X}\,q), \mathcal{F}_\Phi \cup \{q \rightarrow \mathcal{I}_\psi\})$
    end switch
end function
```

Figure 2.6: Recursive algorithm for translating Ltl formulas to non-deterministic generalised Büchi automata (standard translation) [83]. The function NewVar returns a new unused Boolean variable. For an arbitrary Boolean formula φ, $\mathsf{X}\,\varphi$ refers to the formula obtained from φ by swapping all Boolean variables with their primed versions (i.e. switching from the current state variables to the next state variables).



(a) Automaton for the Ltl formula $\mathsf{X}\,p$. There are 2 initial states and no fairness constraints.



(b) Automaton for the Ltl formula $a\,\mathsf{U}\,b$. There are 2 initial states and one fairness constraint (satisfied at nodes with double borders).

Figure 2.7: Automata generated using standard translation. Negated variables are overlined for readability (e.g. $\overline{q_1 q_2}$ stands for $\neg q_1 \wedge \neg q_2$).

in order for q_2 to be true in the first state) or b is infinitely often true (so again, b will be true after a finite number of steps). In both cases, $a\,\mathsf{U}\,b$ holds (thanks to the fact that b will be true for the first time in a finite number of steps, until which point a must be true since q_2 is true in the first state).

$\Leftarrow$: Assume that $a\,\mathsf{U}\,b$ holds on the path. Then b is true at some point in the (finite) future. Moreover, a is true in every state until that point. Hence, by the recursive relation, q_2 is true in the first state.

The resulting generalised Büchi automaton formula equivalent to the Ltl formula $a\,\mathsf{U}\,b$ (obtained by adding the recursive relation and fairness constraint to the automaton product $\mathfrak{A}_a \times \mathfrak{A}'_b$) is:

$$\mathfrak{A}_{a\,\mathsf{U}\,b} = \mathcal{A}_\exists(\{q_1, q_2\}, AP, q_2, (q_1 \leftrightarrow b) \wedge (q_2 \leftrightarrow q_1 \vee a \wedge q'_2), \{q_2 \to q_1\})$$

Note that the exactly same automaton formula is obtained by applying the standard translation. The corresponding automaton is shown in figure 2.7b.

2.4.4 Determinisation

The Büchi[21] automata obtained by the standard translation, discussed in Subsection 2.4.3, are non-deterministic. In order to be able to synthesise strategies which enforce the corresponding Ltl formulas, we need to further transform the non-deterministic automata into equivalent deterministic automata.

Translation of non-deterministic automata to deterministic ω-automata is typically performed using Safra's algorithm [82]. Although the procedure is optimal for Büchi automata, it is very difficult to implement because it generates deterministic Rabin automata over trees, which cannot be efficiently represented symbolically using BDDs [77].

In [77, Section 5], a new determinisation procedure for *non-confluent* Büchi automata, which we will use in our novel model checking algorithm for Sl[1g] in Chapter 5, was proposed. Intuitively, an ω-automaton is non-confluent if every finite prefix of a run is uniquely determined by the last visited state [77].

Definition 2.43 (Non-confluent ω-Automata). Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be an ω-automaton. We say that $\mathfrak{A}$ is *non-confluent* if for every infinite word α we have: if β_1 and β_2 are two runs of $\mathfrak{A}$ on α that intersect at a position t_0 (i.e. $\beta_1(t_0) = \beta_2(t_0)$), then $\beta_1(t) = \beta_2(t)$ for all $t \leq t_0$.

We show how to convert the automata generated by the standard translation to equivalent non-confluent automata before describing the procedure itself. Firstly, we note that the standard translation generates *unambiguous* automata [25]. An automaton is unambiguous if every accepted word has a unique accepting run (although there may be other non-accepting runs).

[21]Standard translation constructs non-deterministic *generalised* Büchi automata. However, as explained in Subsection 2.4.1, it is straightforward to transform a generalised Büchi automaton into an equivalent Büchi automaton.

Definition 2.44 (Unambiguous ω-Automata)**.** Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be an ω-automaton and set $\mathfrak{A}^\star = (S, \Sigma, S, R, \mathcal{F})$. We say that $\mathfrak{A}$ is *unambiguous* iff for every infinite word $\alpha \in \Sigma^\omega$, there is at most one accepting run $\beta \in S^\omega$ of $\mathfrak{A}^\star$ on α[22].

Proposition 2.3. The generalised Büchi automata generated by the standard translation in Figure 2.6 are unambiguous.

It turns out that there is a connection between unambiguous and non-confluent automata: An unambiguous automaton without *useless states* is non-confluent [77]. Intuitively, a state is *not* useless if *(i)* it is reachable from one of the initial states and *(ii)* there exists some accepting run starting from it.

Definition 2.45 (Useless States)**.** Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be an ω-automaton and $q \in S$ a state. We say that q is *useless* if *(i)* there is no initial state $q_I \in I$ such that q is reachable from q_I in $\mathfrak{A}$ or *(ii)* there is no infinite word $\alpha \in \Sigma^\omega$ accepted by the ω-automaton $\mathfrak{A}' = (S, \Sigma, \{q\}, R, \mathcal{F})$.

Proposition 2.4. Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be an *unambiguous* automaton with no useless states. Then $\mathfrak{A}$ is non-confluent.

Therefore, if we remove useless states from an unambiguous automaton, we obtain a non-confluent automaton. Given an automaton $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$, the set of reachable states which are not dead ends (i.e. states which are *not* useless) can be calculated as an intersection of two fixpoints:

$$Q = \underbrace{\mathrm{lfp}_X \left[I \cup \{ s \in S \mid \exists x \in X \exists a \in \Sigma.\, R(x, a, s) \} \right]}_{\text{calculate reachable states}} \cap \underbrace{\mathrm{gfp}_X \{ s \in S \mid \exists x \in X \exists a \in \Sigma.\, R(s, a, x) \}}_{\text{remove dead ends}}$$

This is very simple to implement symbolically using operations on BDDs (see Subsection 2.3.2).

To sum up, we can convert an Ltl formula to an equivalent non-confluent non-deterministic Büchi automaton as follows (all steps can be performed symbolically):

1. We convert the Ltl formula to a generalised Büchi automaton using the standard translation (Subsection 2.4.3). The automaton is unambiguous (Proposition 2.3).
2. We convert the generalised Büchi automaton to a Büchi automaton using automaton product as explained in Subsection 2.4.1. The transformation preserves unambiguity [77].
3. We remove useless states from the Büchi automaton using the fixpoint computation above. The resulting automaton is non-confluent (Proposition 2.4).

Determinisation Procedure for Non-confluent Büchi Automata

A non-confluent Büchi automaton can be determinised using a new procedure presented in [77, 78][23], which is a specialisation of Safra's algorithm for non-confluent automata. Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be a non-confluent Büchi automaton with n reachable states. The procedure constructs a *deterministic parity automaton* $\mathfrak{P}$ equivalent to $\mathfrak{A}$. The states of $\mathfrak{P}$ are $(n+1)$-tuples of pairs (S_i, m_i), where $S_i \subseteq S$ is a set of states and $m_i \in \{\mathtt{0}, \mathtt{1}\}$ a Boolean flag for the visit of accepting states. The initial state of $\mathfrak{P}$ is $((I, \mathtt{0}), (\emptyset, \mathtt{0}), \ldots, (\emptyset, \mathtt{0}))$. To compute the successor of a state $((S_0, m_0), \ldots, (S_n, m_n))$ under input $a \in \Sigma$, we proceed as follows (we treat the tuple as a list):

1. **Delete empty sets.** Let e be the lowest index such that $S_e = \emptyset$. We delete the pair (S_e, m_e) and shift all pairs (S_i, m_i) with $i > e$ to the left. In addition, we delete all pairs from the right end with $S_i = \emptyset$ except for (S_0, m_0).
2. **Calculate successors.** We replace each subset S_i by its existential successors $\mathsf{suc}_\exists^{R,a}(S_i)$:

   $$\mathsf{suc}_\exists^{R,a}(S_i) = \{ s' \in S \mid \exists s \in S_i.\, R(s, a, s') \}$$

   Note that S_0 is the set of states that $\mathfrak{A}$ can reach after having read a finite input word $w \in \Sigma^*$ starting in one of the initial states.

[22]i.e. there is at most one accepting run $\beta \in S^\omega$ of $\mathfrak{A}$ on α starting in an arbitrary state ($\beta(0) \in S$).

[23]The first version of the new procedure was introduced in [78]. The updated version, presented in [77], is slightly different (as explained in the footnote on page 82 in [77]). Our implementation of the updated version did not work properly on some examples, whereas the first one did. Therefore, we describe here the first version. We gained some useful insights about the procedure from [89], where it is tailored for the verification of parametric linear temporal logic (PLtl).

3. **Append.** We append $(S_0 \cap \mathcal{F}, \mathtt{0})$ to the end of the list to remember that the paths leading to states $S_0 \cap \mathcal{F}$ have already visited $\mathcal{F}$.
4. **Mark and clean up.** Let ℓ be the length of the list. For each $i \in \{0, \dots, \ell - 1\}$, if $S_i \setminus \mathcal{F} \subseteq \bigcup_{j=i+1}^{\ell-1} S_j$ and $S_i \neq \emptyset$, then we set $m_i = \mathtt{1}$ and remove S_i from all S_j with $j > i$. Otherwise, we set $m_i = \mathtt{0}$.

   As $\mathfrak{A}$ is non-confluent, we know that a finite run is uniquely characterised by its last state. Hence, if a state occurs in two sets S_i and S_j with $j > i$, then we know that both sets are following the same run. Thus, whenever a state set S_i contains only states which are accepting (i.e. in $\mathcal{F}$) or in any of the state sets S_j with $j > i$, we know that all runs in S_i have visited an accepting state recently. Hence, we also mark S_i as accepting and remove the states S_i from all S_j with $j > i$.
5. **Pad.** Pad the list with $n + 1 - \ell$ pairs $(\emptyset, \mathtt{0})$ at the end. The length of the resulting list is $n + 1$ and hence is a valid state of $\mathfrak{P}$.

   After the clean up, every non-empty state set S_i must contain some state $q \in S_i$ such that $q \notin S_j$ for all $j > i$. Hence, the padded list with $n+1$ pairs must have at least one empty state set. Thus, at least one pair will be removed in the step 1 and the list will again have length at most $n + 1$ after step 3.

We now explain how to determine the colour of an arbitrary state $((S_0, m_0), \dots, (S_n, m_n))$ of $\mathfrak{P}$. Let e be the lowest index such that $S_e = \emptyset$ and m the lowest index i such that $m_i = \mathtt{1}$ (or ∞ if there is no such index). The colouring function $c : S \to \{0, \dots, 2n - 1\}$ is defined as[24]:

$$c(q) = \begin{cases} 1 & \text{if } e = 0 \\ 2m & \text{if } m < e \\ 2e - 1 & \text{if } 0 < e < m \end{cases}$$

The idea of the construction is as follows: If an entry S_i never becomes empty after a certain position on a run π of $\mathfrak{P}$ on $\alpha \in \Sigma^\omega$ and is marked infinitely often, then we know that a run ξ of $\mathfrak{A}$ on α contained in π is introduced infinitely often in step 3. Hence, ξ is accepting[25] and π contains an accepting run of $\mathfrak{A}$. Please refer to Section 3 of [78] for a more detailed explanation. The procedure described above is formalised in the following definition.

Definition 2.46 (Non-confluent Automaton Determinisation)**.** Let $\mathfrak{A} = (S, \Sigma, I, R, \mathcal{F})$ be a non-confluent Büchi automaton with $|S| = n$ states. A deterministic parity automaton $\mathfrak{P} = (T, \Sigma, t_I, \delta, c)$ equivalent to $\mathfrak{A}$ is constructed as follows:

- The states of $\mathfrak{P}$ are $(n + 1)$-tuples of pairs containing a subset of S and a Boolean flag: $T = \{((S_0, m_0), \dots, (S_n, m_n)) \,|\, S_i \subseteq S \wedge m_i \in \{\mathtt{0}, \mathtt{1}\}\}$.
- The initial state is $t_I = ((I, \mathtt{0}), (\emptyset, \mathtt{0}), \dots, (\emptyset, \mathtt{0}))$.
- The transition function δ is defined as follows: Given a state $t = ((S_0, m_0), \dots, (S_n, m_n))$, the next state $t' = ((S'_0, m'_0), \dots, (S'_n, m'_n))$ under input $a \in \Sigma$ satisfies:

$$m'_i = \begin{cases} \left(\mathsf{suc}_\exists^{R,a}(S_i) \setminus \mathcal{F}\right) \subseteq \left(\bigcup_{j=i+1}^{n} \mathsf{suc}_\exists^{R,a}(S_j)\right) & \text{if } i < e \wedge \mathsf{suc}_\exists^{R,a}(S_i) \neq \emptyset \\ \left(\mathsf{suc}_\exists^{R,a}(S_{i+1}) \setminus \mathcal{F}\right) \subseteq \left(\bigcup_{j=i+2}^{n} \mathsf{suc}_\exists^{R,a}(S_j)\right) & \text{if } i \geq e \wedge \mathsf{suc}_\exists^{R,a}(S_{i+1}) \neq \emptyset \\ \mathtt{0} & \text{else} \end{cases}$$

$$S'_0 = \mathsf{suc}_\exists^{R,a}(S_0)$$

$$S'_i = \left(\begin{cases} \mathsf{suc}_\exists^{R,a}(S_i) & \text{if } i < e \\ \mathsf{suc}_\exists^{R,a}(S_{i+1}) & \text{if } e \leq i < E \\ \mathsf{suc}_\exists^{R,a}(S_0) \cap \mathcal{F} & \text{if } e < i = E \vee e = i = E + 1 \\ \emptyset & \text{else} \end{cases} \right) \setminus \left(\bigcup_{\substack{j=0 \\ m'_j = \mathtt{1}}}^{i-1} S'_j \right) \quad \text{for } i > 0$$

[24]Note that $e \leq n$ (at least one state set S_i must be empty) and $m \neq e$ for the reachable states. By considering all three cases in the definition of the colouring function, we see that the maximum possible colour for any reachable state is $2n - 1$ because it cannot be the case that both $m = n$ and $m < e$.

[25]Since the introduction of ξ in step 3 occurs only when an accepting state is encountered.

where e is the lowest index such that $S_e = \emptyset$ and E is the largest index such that $S_E \neq \emptyset$.

- The colour of a state $t = ((S_0, m_0), \ldots, (S_n, m_n))$ is:

$$c(t) = \begin{cases} 1 & \text{if } e = 0 \\ 2m & \text{if } m < e \\ 2e - 1 & \text{if } 0 < e < m \end{cases}$$

where e is the lowest index such that $S_e = \emptyset$ and m is the lowest index i such that $m_i = \mathtt{1}$ (or ∞ if there is no such index).

The complexity of the procedure is provided in the following proposition [77][26]:

Proposition 2.5. Given a non-confluent Büchi automaton with n states, the construction in Definition 2.46 yields a deterministic parity automaton with at most $2^{(n+1)^2}$ states and $2n$ colours.

Examples of the Determinisation Procedure

In order to give some intuition into how the new determinisation procedure works, we will now discuss two examples. Firstly, we consider the non-deterministic unambiguous generalised Büchi automaton $\mathfrak{A}_{\mathsf{X}\,p}$ equivalent to the Ltl formula $\mathsf{X}\,p$ in Figure 2.7a. Since it has no fairness constraints, we can treat it as a non-deterministic unambiguous Büchi automaton where all states are accepting[27]. We start by calculating the set of reachable states which are not dead ends. There are four such states: $s_1 = \overline{q_1}\overline{q_2}$, $s_2 = \overline{q_1}q_2$, $s_3 = q_1\overline{q_2}$, and $s_4 = q_1q_2$. Hence, the states of the equivalent deterministic parity automaton $\mathfrak{P}_{\mathsf{X}\,p}$ will be 5-tuples.

The initial state of the parity automaton is $t_I = ((\{s_2, s_4\}, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}))$. The colour of the initial state is $c(t_I) = 1$ since $e = 1$ and $m = \infty$. In order to determine $\delta(t_I, p)$, we perform steps 1–5 of the determinisation procedure and obtain the next state $t_1 = ((\{s_3, s_4\}, \mathtt{1}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}))$. Also, we find that $\delta(t_I, \overline{p}) = t_1$, i.e. the second state of $\mathfrak{P}_{\mathsf{X}\,p}$ is the same regardless of the input. This is in line with the meaning of $\mathsf{X}\,p$ (only the second input "matters").

The final parity automaton $\mathfrak{P}_{\mathsf{X}\,p}$ is shown in Figure 2.8a. It has four reachable states and uses two colours. It is indeed equivalent to the Ltl formula $\mathsf{X}\,p$: If the second input is $\neg p$, the run will stay forever in the state $((\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}))$ with colour 1. Hence, the minimal colour occurring infinitely often will be odd so the run will not be accepted (see Definition 2.39). Conversely, if the second input is p, the run will stay forever in the state $((\{s_1, s_2, s_3, s_4\}, \mathtt{1}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}), (\emptyset, \mathtt{0}))$ with colour 0. Hence, the minimal colour occurring infinitely often will be even so the run will be accepted. Note that the last four state sets S_1–S_4 are always empty in all reachable states of $\mathfrak{P}_{\mathsf{X}\,p}$. Thus, we could have used single pairs (S_0, m_0) as states of $\mathfrak{P}_{\mathsf{X}\,p}$ (instead of 5-tuples of pairs).

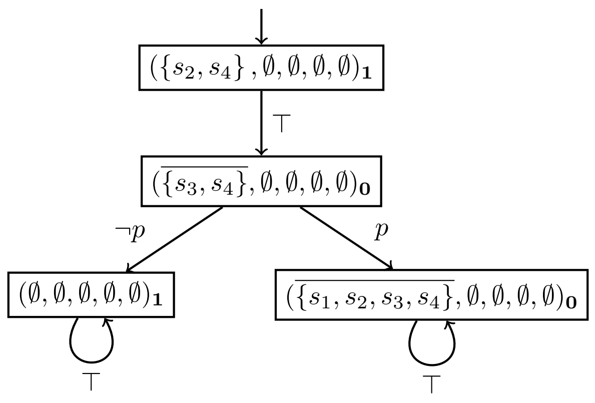
As a second example, we consider the non-deterministic unambiguous generalised Büchi automaton $\mathfrak{A}_{a\,\mathsf{U}\,b}$ equivalent to the Ltl formula $a\,\mathsf{U}\,b$ in Figure 2.7b. Since it has exactly one fairness constraint, we can treat it as a non-deterministic unambiguous Büchi automaton straightaway[28]. There are only three reachable states of $\mathfrak{A}_{a\,\mathsf{U}\,b}$ which are not dead ends: $s_1 = \overline{q_1}\overline{q_2}$, $s_2 = \overline{q_1}q_2$, and $s_3 = q_1q_2$. Hence, the states of the equivalent deterministic parity automaton $\mathfrak{P}_{a\,\mathsf{U}\,b}$ will be 4-tuples. The parity automaton $\mathfrak{P}_{a\,\mathsf{U}\,b}$ is shown in Figure 2.8b. It has eight reachable states and uses three colours. Clearly, a much smaller deterministic parity automaton, also equivalent to $a\,\mathsf{U}\,b$, can be constructed by hand[29]. Note that in this case, all four pairs $(S_0, m_0), \ldots, (S_3, m_3)$ are required in the tuple.

[26]Note that since we use the original version of the procedure introduced in [78], the tuples consist of $n+1$ pairs (S_i, m_i) (rather than n as in the updated version [77]). There are 2 possible values for m_i and 2^n possible values for each S_i. Hence, there are $(2 \times 2^n)^{n+1} = 2^{(n+1)^2}$ possible states of the automaton. We also provide a slightly tighter bound on the number of colours because we have shown that the maximum possible colour of any reachable state is $2n - 1$.

[27]If we apply the general procedure for converting generalised Büchi automata to Büchi automata in Subsection 2.4.1, the auxiliary deterministic Büchi automaton will be $\mathfrak{D} = (\{q_a\}, S, q_a, \delta, \{q_a\})$ with $\delta(q_a, s) = q_a$ for all $s \in S$. Observe that $\mathfrak{D}$ has only one state, which is accepting, and accepts all infinite paths $\pi \in S^\omega$. Hence, the automaton product will have identical states and transitions as the original generalised Büchi automaton. Moreover, all its states will be accepting.

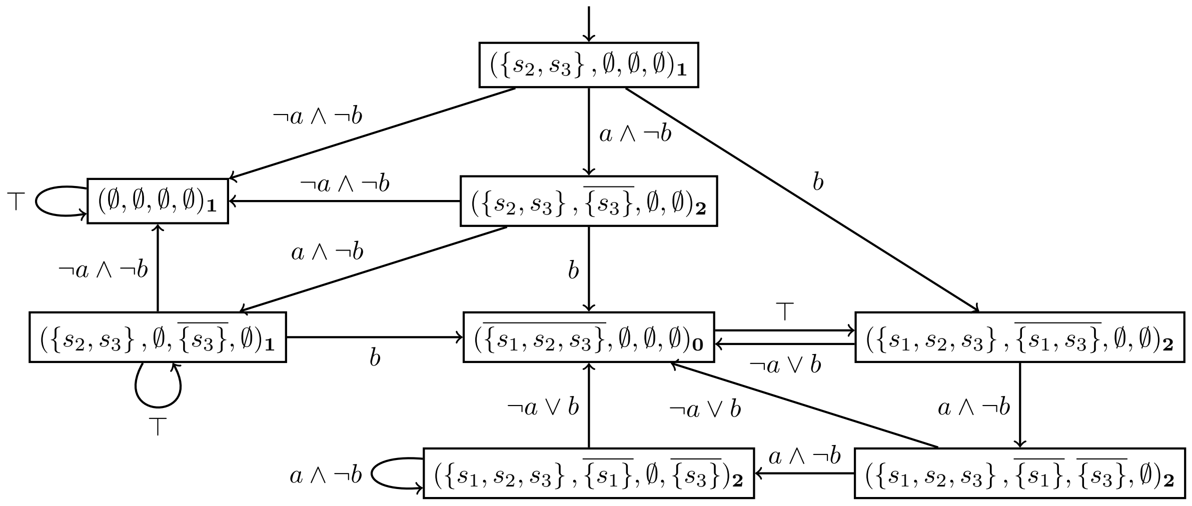
[28]The general procedure for converting generalised Büchi automata to Büchi automata in Subsection 2.4.1 would actually double the number of states. This is due to the fact that the auxiliary deterministic Büchi automaton $\mathfrak{D}$ would have two states.

[29]However, constructing automata manually is only feasible for simple Ltl formulas. For a more systematic approach, please refer to Chapter 3 of [77] for an introduction to minimisation of ω-automata.



$(\{s_2, s_4\}, \emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{1}}$
$\top$
$(\overline{\{s_3, s_4\}}, \emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{0}}$
$\neg p$
p
$(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{1}}$
$(\overline{\{s_1, s_2, s_3, s_4\}}, \emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{0}}$
$\top$
$\top$

(a) Deterministic parity automaton for the LTL formula $\mathsf{X}\, p$.



$(\{s_2, s_3\}, \emptyset, \emptyset, \emptyset)_{\mathbf{1}}$
$\neg a \wedge \neg b$
$a \wedge \neg b$
b
$\top$
$(\emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{1}}$
$\neg a \wedge \neg b$
$(\{s_2, s_3\}, \overline{\{s_3\}}, \emptyset, \emptyset)_{\mathbf{2}}$
$\neg a \wedge \neg b$
$a \wedge \neg b$
b
$(\{s_2, s_3\}, \emptyset, \overline{\{s_3\}}, \emptyset)_{\mathbf{1}}$
b
$(\overline{\{s_1, s_2, s_3\}}, \emptyset, \emptyset, \emptyset)_{\mathbf{0}}$
$\top$
$\neg a \vee b$
$(\{s_1, s_2, s_3\}, \overline{\{s_1, s_3\}}, \emptyset, \emptyset)_{\mathbf{2}}$
$\top$
$\neg a \vee b$
$\neg a \vee b$
$a \wedge \neg b$
$a \wedge \neg b$
$(\{s_1, s_2, s_3\}, \overline{\{s_1\}}, \emptyset, \overline{\{s_3\}})_{\mathbf{2}}$
$a \wedge \neg b$
$(\{s_1, s_2, s_3\}, \overline{\{s_1\}}, \overline{\{s_3\}}, \emptyset)_{\mathbf{2}}$

(b) Deterministic parity automaton for the LTL formula $a \,\mathsf{U}\, b$.

Figure 2.8: Deterministic parity automata constructed using the new determinisation procedure. Marked sets are overlined for readability (e.g. $\overline{\{s_3, s_4\}}$ stands for $(\{s_3, s_4\}, \mathtt{1})$). The colours of the states are written as subscripts (e.g. the colour of the state $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)_{\mathbf{1}}$ is 1).

Symbolic Implementation of the Determinisation Procedure

Although the determinisation procedure is quite complicated, it is possible to implement it symbolically [77, 78]. Let $\mathfrak{A} = \mathcal{A}_\exists(V, AP, \mathcal{I}, \mathcal{R}, \mathcal{F})$ be a non-deterministic non-confluent Büchi automaton with m state variables $v_0, \ldots, v_{m-1} \in V$ and n reachable states[30] $s_0, s_1, \ldots, s_{n-1} \in S \subseteq 2^V$. Hence, the states of the equivalent parity automaton $\mathfrak{P} = \mathcal{A}_\exists\left(Q, AP, \mathcal{I}_P, \mathcal{R}_P, \langle \mathcal{C}_i \rangle_{i \in \{0,\ldots,2n+1\}}\right)$ are $(n+1)$-tuples $((S_0, m_i), \ldots, (S_n, m_n))$. For each subset S_i, we allocate n variables $q_i^0, \ldots q_i^{n-1}$ such that q_i^k represents that $s_k \in S_i$. Furthermore, we introduce $n+1$ variables $m_0, \ldots, m_n$ to represent the markings. The set of state variables of $\mathfrak{P}$ is thus:

$$Q \triangleq \left\{q_i^k \;\middle|\; 0 \leq k < n \wedge 0 \leq i \leq n\right\} \cup \{m_i \mid 0 \leq i \leq n\}$$

In order to be able to express the transition function, we also need the next versions of all variables in Q (i.e. ${q_i^k}'$ and ${m_i}'$). The initial state of $\mathfrak{P}$ is encoded as:

$$\mathcal{I}_P \triangleq \underbrace{\left[\bigwedge_{s_k \in \mathcal{I}} q_0^k\right] \wedge \left[\bigwedge_{s_k \notin \mathcal{I}} \neg q_0^k\right]}_{S_0 = \mathcal{I}} \wedge \underbrace{\left[\bigwedge_{i=1}^{n} \bigwedge_{k=0}^{n-1} \neg q_i^k\right]}_{S_i = \emptyset \text{ for all } i>0} \wedge \underbrace{\left[\bigwedge_{i=0}^{n} \neg m_i\right]}_{m_i = 0 \text{ for all } i}$$

Before encoding the transition relation, we need to define several auxiliary formulas. Firstly, we need to express the successor function $\mathsf{suc}_\exists^{\mathcal{R},a}(S_i)$. For this purpose we introduce a family of formulas φ_i^k over $Q \cup AP$, such that φ_i^k is true iff $s_k \in \mathsf{suc}_\exists^{\mathcal{R},a}(S_i)$:

$$\varphi_i^k \triangleq \bigvee_{j=0}^{n-1} q_i^j \wedge \underbrace{\exists v_0 \ldots v_{m-1} v_0' \ldots v_{m-1}'. s_j \wedge \mathcal{R} \wedge s_k'}_{\substack{\text{propositional formula over } AP \\ \text{for transition from } s_j \text{ to } s_k}}$$

Secondly, we need to express the indices e and E. We introduce a family of formulas $\Gamma_{i \geq e}$ and $\Gamma_{i=E}$, such that Γ_ε is true iff the expression ε is true:

$$\Gamma_{i \geq e} \triangleq \underbrace{\bigvee_{j=0}^{i} \bigwedge_{k=0}^{n-1} \neg q_j^k}_{S_j = \emptyset \text{ for some } j \leq i} \qquad\qquad \Gamma_{i=E} \triangleq \underbrace{\left[\bigvee_{k=0}^{n-1} q_i^k\right]}_{S_i \neq \emptyset} \wedge \underbrace{\left[\bigwedge_{j=i+1}^{n} \bigwedge_{k=0}^{n-1} \neg q_j^k\right]}_{S_j = \emptyset \text{ for all } j>i}$$

We are now ready to encode the transition relation. Recall that a state set S_i is marked (i.e. $m_i = 1$) iff it is non-empty and every non-accepting state $s_k \in S_i \setminus \mathcal{F}$ appears in some state set S_j with $j > i$. Thus, the equation for the next value of each of the variables m_i is:

$$\Xi_i \triangleq m_i' \leftrightarrow \underbrace{\left[\bigvee_{k=0}^{n-1} {q_i^k}'\right]}_{S_i' \neq \emptyset} \wedge \underbrace{\left[\bigwedge_{s_k \in S \setminus \mathcal{F}} \left({q_i^k}' \rightarrow \bigvee_{j=i+1}^{n} \varphi_j^k\right)\right]}_{(S_i' \setminus \mathcal{F}) \subseteq (\bigcup_{j=i+1}^{n} S_j')}$$

There are three possible cases for the next state set S_i': *(i)* successors of the local state set $\mathsf{suc}_\exists^{\mathcal{R},a}(S_i)$ (when $i < e$), *(ii)* successors of the right state set $\mathsf{suc}_\exists^{\mathcal{R},a}(S_{i+1})$ (when $e \leq i < E$), or *(iii)* accepting successors of the first state set $\mathsf{suc}_\exists^{\mathcal{R},a}(S_0) \cap \mathcal{F}$ (when $e < i = E$ or $e = i = E + 1$). If none of the conditions is satisfied, $S_i' = \emptyset$. The first case always applies to the first set S_0' (regardless of the values of e and E):

$$\Delta_0^k \triangleq {q_0^k}' \leftrightarrow \varphi_0^k$$

[30]Note that although the set of reachable states can be calculated symbolically, we need to explicitly enumerate them. Therefore, strictly speaking, the procedure is *semi-symbolic*.

For $i > 0$, when a state $s_k \notin \mathcal{F}$ is not accepting, the first two cases apply to S_i':

$$\Phi_i^k \triangleq {q_i^k}' \leftrightarrow (\underbrace{\neg\Gamma_{i\geq e} \wedge \varphi_i^k}_{\text{first case}} \vee \underbrace{\Gamma_{i\geq e} \wedge \varphi_{i+1}^k}_{\text{second case}}) \wedge \underbrace{\neg \bigvee_{j=0}^{i-1} \left({q_j^k}' \wedge m_j'\right)}_{\substack{\text{remove marked state}\\ \text{sets on the left}}}$$

For $i > 0$, when a state $s_k \in \mathcal{F}$ is accepting, all three cases apply to S_i':

$$\Psi_i^k \triangleq {q_i^k}' \leftrightarrow (\neg\Gamma_{i\geq e} \wedge \varphi_i^k \vee \Gamma_{i\geq e} \wedge \varphi_{i+1}^k \vee \underbrace{[\overbrace{\Gamma_{i\geq e} \wedge \Gamma_{i=E}}^{e<i=E} \vee \overbrace{\neg\Gamma_{i-1\geq e} \wedge \Gamma_{i-1=E}}^{e=i=E+1}] \wedge \varphi_0^k}_{\text{third case}}) \wedge \neg \bigvee_{j=0}^{i-1} \left({q_j^k}' \wedge m_j'\right)$$

Finally, the complete transition relation is:

$$\mathcal{R}_P \triangleq \underbrace{\left[\bigwedge_{k=0}^{n-1} \Delta_0^k \wedge \Xi_0\right]}_{(S_0', m_0')} \wedge \bigwedge_{i=1}^{n} \underbrace{\left[\left(\bigwedge_{s_k \notin \mathcal{F}} \Phi_i^k\right) \wedge \left(\bigwedge_{s_k \in \mathcal{F}} \Psi_i^k\right) \wedge \Xi_i\right]}_{(S_i', m_i') \text{ for all } i>0}$$

It remains to show how to encode the colouring function. We represent it by a family of formulas $\mathcal{C}_0, \ldots, \mathcal{C}_{2n+1}$ such that a state $t \in 2^Q$ has colour c iff the expression $\exists q_0 \ldots q_{|Q|-1}.\mathcal{C}_c \wedge t$ is true. Recall, that the colouring function refers to the lowest indices e and m such that $S_i = \emptyset$ and $m_i = \mathtt{1}$, respectively:

$$\Gamma_{i=e} \triangleq \Gamma_{i\geq e} \wedge \neg\Gamma_{i-1\geq e} \qquad \Gamma_{i\geq m} \triangleq \bigvee_{j=0}^{i} m_j \qquad \Gamma_{i=m} \triangleq \Gamma_{i\geq m} \wedge \neg\Gamma_{i-1\geq m}$$

The formulas $\mathcal{C}_{2k}$ and $\mathcal{C}_{2k-1}$ are defined as follows (note that $\mathcal{C}_1$ has an extra term for the special case $e = 0$):

$$\begin{aligned} \mathcal{C}_1 &\triangleq \Gamma_{1=e} \wedge \neg\Gamma_{0\geq m} \vee \Gamma_{0=e} & \\ \mathcal{C}_{2k-1} &\triangleq \Gamma_{k=e} \wedge \neg\Gamma_{k-1\geq m} & \text{with } 2 \leq k \leq n+1 \\ \mathcal{C}_{2k} &\triangleq \Gamma_{k=m} \wedge \neg\Gamma_{k\geq e} & \text{with } 0 \leq k \leq n \end{aligned}$$

This completes the symbolic implementation of the determinisation procedure.

When considering the deterministic parity automaton $\mathfrak{P}_{\mathsf{X}p}$ (shown in Figure 2.8a) equivalent to the LTL formula $\mathsf{X}p$, we found that most often not all pairs (S_i, m_i) are necessary. As suggested in [77], we can construct the parity automaton for a fixed bound $b \leq n+1$ (i.e. use b-tuples instead of $(n+1)$-tuples) and check if the bound is sufficient: We calculate the set of reachable states and check if any of them requires more than b pairs under some input. More formally, let X be a formula representing the set of reachable states when using b-tuples. The bound is *sufficient* iff the following logical expression is equivalent to $\bot$ (essentially, we are testing the third case in Ψ_i^b):

$$X \wedge \mathcal{R}_P \wedge \neg\Gamma_{b-1\geq e} \wedge \Gamma_{b-1=E} \wedge \bigvee_{s_k \in \mathcal{F}} \left[\varphi_0^k \wedge \neg \bigvee_{j=0}^{b-1} \left({q_j^k}' \wedge m_j'\right)\right]$$

If the bound is not sufficient, we increase it and try again. It can be shown that the range of the colouring function on reachable states for a sufficient bound b is $\{0, \ldots, \min(2b-1, 2n-1)\}$[31].

[31]We have already explained why the colour cannot be greater than $2n-1$. If the bound b is sufficient, then the state set S_b would be empty in all reachable states (that is exactly why we did not increase the bound any further). Hence, we have $e \leq b$ in all reachable states. By considering all three cases in the definition of the colouring function, we see that the maximum colour cannot exceed $2b-1$.

2.4.5 Games

As we have mentioned already, many problems in the area of software verification can be reduced to solving infinite two-player games. In fact, our novel SL[1G] model checking algorithm presented in Chapter 5 relies on such a reduction. Therefore, we give a brief overview of infinite two-player games on directed graphs in this subsection. Please refer to [68], which this subsection is largely based on, for a more detailed introduction.

A game consists of an arena and a winning set[32]. Intuitively, an *arena* represents the "board" on which a game is played. A formal definition follows.

Definition 2.47 (Arenas)**.** Let V_0 and V_1 be sets of *0-vertices* and *1-vertices*, which are disjoint (i.e. $V_0 \cap V_1 = \emptyset$). Furthermore, let $E \subseteq V \times V$ be an *edge relation*, where $V = V_0 \cup V_1$ is the set of all vertices. Then an *arena* is a triple $\mathcal{A} = (V_0, V_1, E)$.

The set of *successors* of a vertex $v \in V$ is defined as $vE \triangleq \{v' \in V \mid (v, v') \in E\}$. A vertex $\overline{v} \in V$ is a *dead end* iff $\overline{v}E = \emptyset$ (i.e. it has no successors).

The games we are considering have only two players, player 0 and player 1. We often talk about player σ, where $\sigma \in \{0, 1\}$. The other player is referred to as player $\overline{\sigma}$ (i.e. $\overline{\sigma} = 1 - \sigma$).

The game is played as follows: A token is initially placed on some vertex $v \in V$. If it is a 0-vertex ($v \in V_0$), player 0 moves the token to one of its successors vE. Symmetrically, if it is a 1-vertex ($v \in V_1$), player 1 moves the token to one of its successors vE. This pattern is repeated forever or until the current vertex v is a dead end. Such finite and infinite sequences of vertices are called *plays*[33].

Definition 2.48 (Game Plays)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena. Then a *play* in $\mathcal{A}$ is either:

- a non-empty finite path $\pi = v_0 v_1 \dots v_n \in V^+$ such that $v_i \in v_{i-1}E$ for all $0 < i \leq n$ and $v_n E = \emptyset$ (*finite play*) or
- an infinite path $\pi = v_0 v_1 \dots \in V^\omega$ such that $v_i \in v_{i-1}E$ for all $i > 0$ (*infinite play*).

Having defined an arena and a play, we are ready to define a *game* and the *winner of a play*. Intuitively, player 0 wins a game if he forces player 1 into a dead end or if the infinite play satisfies some winning condition.

Definition 2.49 (Games)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena and $Win \subseteq V^\omega$ a *winning set*. Then the pair $\mathfrak{G} = (\mathcal{A}, Win)$ is a *game*.

Definition 2.50 (Winner of a Play)**.** Let $\mathfrak{G} = (\mathcal{A}, Win)$ be a game and π a play in $\mathcal{A}$. Then player 0 is the *winner* of π in $\mathfrak{G}$ iff:

- π is a finite play $\pi = v_0 v_1 \dots v_n$ where v_n is a 1-vertex ($v_n \in V_1$) and a dead end ($v_n E = \emptyset$) or
- π is an infinite play and $\pi \in Win$.

Otherwise, player 1 is the winner of π in $\mathfrak{G}$.

Similarly to acceptance conditions for ω-automata (see Definition 2.39), there is a multitude of winning conditions for games with different expressive power and translation complexity. In addition, the conditions differ in the amount of memory that a player needs in order to be able to win. Here we present only the basic winning conditions [68].

Definition 2.51 (Winning Conditions)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena. The winning set Win of the game $\mathfrak{G} = (\mathcal{A}, Win)$ is the set of all infinite plays $\pi \in V^\omega$ in $\mathcal{A}$ such that:

- **Büchi winning condition** $F \subseteq V$ (a set of vertices):

$$\mathrm{Inf}(\pi) \cap F \neq \emptyset$$

[32]Notice the analogy with ω-automata (Definition 2.40), which consist of a semi-automaton and an acceptance condition.

[33]We also refer to *plays* in the context of SL semantics (see Definition 2.26). It should always be clear from the context which concept we are referring to.

- **Muller winning condition** $\mathcal{F} \subseteq 2^V$ (a set of vertex sets):
$$\mathrm{Inf}(\pi) \in \mathcal{F}$$
- **Rabin winning condition** $\Omega = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ where $E_i, F_i \subseteq V$ (a family of pairs of vertex sets):
$$\exists i \in \{1, \ldots, k\}.\ (\mathrm{Inf}(\pi) \cap E_i = \emptyset) \wedge (\mathrm{Inf}(\pi) \cap F_i \neq \emptyset)$$
- **Streett winning condition** $\Omega = \{(E_1, F_1), \ldots, (E_k, F_k)\}$ where $E_i, F_i \subseteq V$ (a family of pairs of vertex sets):
$$\forall i \in \{1, \ldots, k\}.\ (\mathrm{Inf}(\pi) \cap E_i \neq \emptyset) \vee (\mathrm{Inf}(\pi) \cap F_i = \emptyset)$$
- **Parity winning condition** $c : V \to \{0, \ldots, k\}$ with $k \in \mathbb{N}$ (a colouring function):
$$\min \{c(v) \mid v \in \mathrm{Inf}(\pi)\} \text{ is even}$$
- **Generalised parity winning condition** $c_1, \ldots, c_n : V \to \{0, \ldots, k_i\}$ with $k_i \in \mathbb{N}$ (a family of colouring functions):
$$\forall i \in \{1, \ldots, n\}. \min \{c_i(v) \mid v \in \mathrm{Inf}(\pi)\} \text{ is even}$$

Notice the correspondence between winning conditions for games and acceptance conditions for ω-automata (see Definition 2.39). We shall use this observation in Chapter 5 to combine an ω-automaton and an arena into a two-player game game (see Definition 5.7).

Given a game $\mathfrak{G}$, we usually want to find out which states of the underlying arena are winning for each player (i.e. from which states a player can force a winning condition to be true). The set of all such states is referred to as the *winning region.* Moreover, we want to construct the corresponding *winning strategies.* Intuitively, a strategy[34] is a function which maps histories of the game (finite prefixes of plays) to moves (next vertices).

Definition 2.52 (Game Strategy)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena and $\sigma \in \{0, 1\}$ a player index. Then a *strategy* for player σ is a partial function $f_\sigma : V^*V_\sigma \rightharpoonup V$, such that for all $\pi \in V^*$ and $v \in V_\sigma$, if $\pi v \in \mathrm{dom}(f_\sigma)$, then $(v, f_\sigma(\pi v)) \in E$.

A prefix of a play $\pi = v_0 v_1 \ldots v_n$ is *conform* with f_σ iff for every $0 < i \leq n$, if $v_0 v_1 \ldots v_{i-1} \in \mathrm{dom}(f_\sigma)$, then $v_i = f_\sigma(v_0 v_1 \ldots v_{i-1})$. A play is conform with f_σ iff each of its prefixes is conform with f_σ.

Definition 2.53 (Winning Strategies)**.** Let $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ be a game, $U \subseteq V$ a set of vertices, and f_σ a strategy for player σ. f_σ is a *winning strategy* for player σ on U iff all plays conform with f_σ starting in U are winning for player σ.

Definition 2.54 (Winning Regions)**.** Let $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ be a game. The winning region $W_\sigma(\mathfrak{G}) \subseteq V$ (or W_σ for short) is the set of all vertices $v \in V$, such that player σ has a winning strategy f_σ on $\{v\}$.

Clearly, given a game $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ the winning regions $W_0(\mathfrak{G})$ and $W_1(\mathfrak{G})$ are always disjoint. If it is also the case that $W_0(\mathfrak{G})$ and $W_1(\mathfrak{G})$ form a partition of V (i.e. $W_0(\mathfrak{G}) \cup W_0(\mathfrak{G}) = V$), then the game is *determined.* Intuitively, a game is determined iff each vertex of the underlying arena is winning for one of the players. It turns out that all the winning conditions in Definition 2.51 have this property [68].

Definition 2.55 (Determinacy)**.** Let $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ be a game with winning regions $W_0(\mathfrak{G})$ and $W_1(\mathfrak{G})$. We say that $\mathfrak{G}$ is *determined* iff $\{W_0(\mathfrak{G}), W_1(\mathfrak{G})\}$ is a partition of V.

Proposition 2.6. All winning conditions in Definition 2.51 are determined.

The definition of a strategy is very general. In particular, it refers to the whole history of the game, which is finite but arbitrarily long. Thus, in theory, a player might need infinite memory to implement such a strategy. Clearly, this is not practical. Therefore, we consider two special types of strategies: *finite memory strategies* (also called *forgetful*) and *memoryless strategies* (also called *positional*).

[34]Again, we also refer to *strategies* in the context of Atl and Sl (see Definitions 2.16 and 2.21). The main difference is that Atl/Sl strategies map sequences of states to actions, whereas game strategies map sequences of states to next states. It should be clear from the context which of the concepts we are referring to.

winning condition	player 0	player 1
reachability	yes	yes
Büchi	yes	yes
Müller	no	no
Rabin	yes	no
Streett	no	yes
parity	yes	yes
generalised parity	no	yes

Table 2.4: Memoryless determinacy of various infinite games [28, 68]. "yes" means that the player wins every game with the corresponding winning condition using a memoryless strategy.

Definition 2.56 (Finite Memory Strategies)**.** Let $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ be a game. A strategy f_σ is *finite memory* if there exists a finite set M, an element $m_I \in M$, and partial functions $\delta : V \times M \rightharpoonup M$ and $g : V_\sigma \times M \rightharpoonup V$ such that the following holds: For each prefix of a play $\pi = v_0 v_1 \dots v_n$ and sequence $m_0 m_1 \dots m_n$ where *(i)* $m_0 = m_I$ and *(ii)* $(v_i, m_i) \in \mathrm{dom}(\delta)$ implies $m_{i+1} = \delta(v_i, m_i)$ for all $0 \leq i < n$, we have either *(a)* $\pi \notin \mathrm{dom}(f_\sigma)$ and $(v_n, m_n) \notin \mathrm{dom}(g)$, or *(b)* $\pi \in \mathrm{dom}(f_\sigma)$, $(v_n, m_n) \in \mathrm{dom}(g)$, and $f_\sigma(\pi) = g(v_n, m_n)$.

Intuitively, the set M represents the possible memory states of player σ and m_I is his initial memory state. The strategy is executed by player σ as follows:

Let the current vertex be v_i and the current memory state of player σ m_i.

1. **Move.** If $v_i \in V_\sigma$ and $(v_i, m_i) \in \mathrm{dom}(g)$, then move to $v_{i+1} = g(v_i, m_i)$. Otherwise, $v_{i+1} \in v_i E$ is arbitrary or selected by player $\overline{\sigma}$.
2. **Update memory.** If $(v_i, m_i) \in \mathrm{dom}(\delta)$, update the memory state of player σ to $m_{i+1} = \delta(v_i, m_i)$. Otherwise, $m_{i+1} \in M$ is arbitrary. Player σ can now forget m_i. Note that the update is carried out even when the move is decided by player $\overline{\sigma}$.

If we set M to be the singleton set $\{m_I\}$, then the strategy becomes memoryless. Intuitively, when using a memoryless strategy, the next vertex depends only on the current vertex. Hence, we can view it as a partial function $V_\sigma \rightharpoonup V$.

Definition 2.57 (Memoryless Strategy)**.** Let $\mathfrak{G} = (\mathcal{A}, \mathit{Win})$ be a game. A strategy f_σ is *memoryless* iff for every prefixes of plays $\pi_1 v, \pi_2 v \in V^* V_\sigma$, we have either *(i)* $\pi_1 v \notin \mathrm{dom}(f_\sigma)$ and $\pi_2 v \notin \mathrm{dom}(f_\sigma)$, or *(ii)* $\pi_1 v \in \mathrm{dom}(f_\sigma)$, $\pi_2 v \in \mathrm{dom}(f_\sigma)$, and $f_\sigma(\pi_1 v) = f_\sigma(\pi_2 v)$.

Finite memory is often sufficient for winning strategies in infinite games. In fact, all games considered in this thesis have this property, referred to as *finite memory determinacy* [68]. Furthermore, some games require no memory at all (so-called *memoryless determinacy*). These include reachability and parity games [68], both of which will be discussed shortly. Table 2.4 shows which players always have a memoryless winning strategy for various winning conditions.

Proposition 2.7. In every game with a winning condition from Definition 2.51, both players win using finite memory strategies.

Proposition 2.8. In every reachability and parity game, both players win using memoryless strategies.

We will now discuss solving reachability games, parity games, and generalised parity games in more detail, as they are relevant for the new model checking algorithm for SL[1G] we propose in Chapter 5.

Reachability Games

We start by considering *reachability games* [68]. We will use them to demonstrate some concepts which will be useful for solving more complex games later. By *solving* we mean finding the winning regions and strategies for both players.

In reachability games, the aim (of player 0) is simply to reach some vertex of a designated set of vertices (or force player 1 into a dead end).

Definition 2.58 (Reachability Game)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena and $X \subseteq V$ a set of vertices. Then a *reachability game* $R(\mathcal{A}, X)$ is a game in which a play π is winning iff *(i)* some vertex from X occurs in π or *(i)* a dead end for player 1 occurs in π.

Note that this differs from all games studied so far because a finite path ending in a dead end for player 0 can still be winning for player 0 as long as some vertex from X is visited. The game is solved by calculating the *attractor* of X for player 0. Intuitively, an attractor of a set of vertices X for player σ is the set of all vertices from which player σ can force a visit of X (regardless of what player $\overline{\sigma}$ does).

Definition 2.59 (σ-Attractor)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena and $X \subseteq V$ a set of vertices. The *σ-attractor* of X in $\mathcal{A}$ is the set of vertices $\mathsf{attr}_\sigma(\mathcal{A}, X) \triangleq \bigcup_{i \geq 0} X^i$ where:

$$\begin{aligned} X^0 &\triangleq X \\ X^{i+1} &\triangleq \left\{ v \in V_\sigma \;\middle|\; vE \cap X^i \neq \emptyset \right\} \cup \left\{ v \in V_{\overline{\sigma}} | vE \subseteq X^i \right\} \end{aligned}$$

A (memoryless) *attractor strategy* for player σ is a partial function $f_\sigma : V_\sigma \rightharpoonup V$ such that $\mathrm{dom}(f_\sigma) = V_\sigma \cap (\mathsf{attr}_0(X) \setminus X)$ and, for each vertex $v \in X^{i+1} \cap V_\sigma$, we have $f_\sigma(v) \in X^i$.

Intuitively, the attractor strategy ensures that the play is moving "towards" X. On the other hand, player σ is "trapped" inside $V \setminus \mathsf{attr}_\sigma(\mathcal{A}, X)$, i.e. player $\overline{\sigma}$ can ensure that the game stays outside $\mathsf{attr}_\sigma(\mathcal{A}, X)$ forever. Such a set of vertices is referred to as a *trap*.

Definition 2.60 (σ-Trap)**.** Let $\mathcal{A} = (V_0, V_1, E)$ be an arena. A set of vertices $Y \subseteq V$ is a *σ-trap* iff *(i) $vE \subseteq Y$* for every $v \in Y \cap V_\sigma$ and *(ii) $vE \cap Y \neq \emptyset$* for every $v \in Y \cap V_{\overline{\sigma}}$. A (memoryless) *trapping strategy* for player $\overline{\sigma}$ is a partial function $f_{\overline{\sigma}} : V_{\overline{\sigma}} \rightharpoonup V$ such that $\mathrm{dom}(f_{\overline{\sigma}}) = Y \cap V_{\overline{\sigma}}$ and, for each vertex $v \in Y \cap V_{\overline{\sigma}}$, we have $f_{\overline{\sigma}}(v) \in V_{\overline{\sigma}}$.

Notice that all σ-attractors (including $\mathsf{attr}_\sigma(\mathcal{A}, \emptyset)$) contain all dead ends for player $\overline{\sigma}$ (since $vE = \emptyset \subseteq X^i$). Hence, the winning regions of a reachability game $R(\mathcal{A}, X)$ are:

$$(W_0, W_1) = (\mathsf{attr}_0(\mathcal{A}, X), V \setminus \mathsf{attr}_0(\mathcal{A}, X))$$

The winning strategies for the players are as follows: Player 0 uses an attractor strategy in his winning region $W_0 = \mathsf{attr}_0(\mathcal{A}, X)$, i.e. he ensures that X will be reached eventually. Conversely, player 1 uses a trapping strategy in this winning region $W_1 = V \setminus \mathsf{attr}_0(\mathcal{A}, X)$, through which he ensures that the attractor of X will never be visited. The following proposition follows from the fact that the attractor of any set in a finite arena with n vertices and m vertices can be calculated in time $O(m + n)$ [68]:

Proposition 2.9. Let $\mathcal{A} = (V_0, V_1, E)$ be an arena with $n = |V| = |V_0| + |V_1|$ vertices and $m = |E|$ edges. The reachability game $R(\mathcal{A}, X)$ for an arbitrary set of vertices $X \subseteq V$ can be calculated in time $O(m + n)$.

Parity Games

The parity winning condition is the "most fundamental" winning condition [46]. It has the following desirable properties:

- Every other winning condition for two-player infinite games can be reduced to it [68].
- It enjoys memoryless determinacy (Proposition 2.8).
- It is easily dualisable[35]. The dual of a parity winning condition c is again a parity winning condition c^{D} with $c^{\mathrm{D}}(v) \triangleq c(v) + 1$.

Recall that player 0 wins a play π in a parity game $\mathfrak{G} = (\mathcal{A}, c)$ iff either *(i)* the minimum[36] colour occurring infinitely often along π is even, or *(ii)* π leads to a dead end for player 1. The solution of a parity game is obtained recursively by solving subgames first and then combining the subresults. A subgraph and a subgame are defined as follows.

[35]Let $\mathfrak{G} = ((V_0, V_1, E), \mathit{Win})$ be a game. The *dual* of $\mathfrak{G}$ is a game $\mathfrak{G}^{\mathrm{D}} \triangleq ((V_1, V_0, E), V^\omega \setminus \mathit{Win})$, i.e. the roles of the players are swapped and the winning condition is complemented.

[36]Equivalently, the parity winning condition is sometimes defined with respect to the *maximum* colour occurring infinitely often.

Definition 2.61 (Subgraph and Subgame)**.** Let $\mathfrak{G} = (\mathcal{A}, c)$ be a parity game with arena $\mathcal{A} = (V_0, V_1, E)$ and $U \subseteq V$ a set of vertices. The *subgraph* of $\mathfrak{G}$ induced by U is defined as $\mathfrak{G}[U] \triangleq (\mathcal{A}|_U, c|_U)$, where $\mathcal{A}|_U \triangleq (V_0 \cup U, V_1 \cup U, E \cup (U \times U))$, $\mathrm{dom}(c|_U) \triangleq U$, and $c|_U(u) \triangleq c(u)$ for all $u \in U$.
A subgraph $\mathfrak{G}[U]$ is a *subgame* of $\mathfrak{G}$ iff every dead end in $\mathfrak{G}[U]$ is also a dead end in $\mathfrak{G}$.

The algorithm for solving parity games shown in Figure 2.9 is originally due to McNaughton and Zielonka [70, 88]. It returns the winning regions of the game and can also be used to construct the corresponding winning strategies. It proceeds as follows:

1. **Case 1.** If the game has no vertices, the algorithm simply returns empty regions (line 3).
2. The algorithm finds the lowest colour among all vertices p, the associated player σ ($\sigma = 0$ if p is even, $\sigma = 1$ if p is odd), and the set of all such vertices X. Then it calculates their attractor A.
3. The algorithm recursively finds the winning regions (W_0, W_1) of the subgame of $\mathfrak{G}$ with all vertices in A removed (line 9).
4. **Case 2.** If no vertices of the subgame are winning for player $\overline{\sigma}$ (line 10), then we are done. Player σ will either keep winning in the subgame from some point onwards, or player $\overline{\sigma}$ can infinitely often move out of the subgame into the attractor A, in which case player σ can ensure that nodes with priority p are visited infinitely often. Hence, all vertices of the game are winning for player σ. The winning strategy is the combination of the winning strategy for the subgame and the attractor strategy for A.

   For strategy synthesis, it is important to ensure that the moves made by player σ in vertices X stay within $\mathfrak{G}$. Remember that $\mathfrak{G}$ might be a subgame of a larger game that we are trying to solve.
5. Since $V \setminus A$ is a σ-trap in the game (otherwise, player σ could reach A from some vertex $v \in V \setminus A$ and since A is an attractor we would have $v \in A$, which is a contradiction), we have $W_{\overline{\sigma}}(\mathfrak{G}[V \setminus A]) \subseteq W_{\overline{\sigma}}(\mathfrak{G})$. Thus, the algorithm calculates the attractor B of $W_{\overline{\sigma}}$ (line 13). Clearly, B is a subset of the winning region for player $\overline{\sigma}$ in $\mathfrak{G}$.
6. The algorithm recursively finds the winning regions (W_0, W_1) of the subgame of $\mathfrak{G}$ with all vertices in B removed (line 14).
7. **Case 3.** We have $W_\sigma(\mathfrak{G}) = W_\sigma(\mathfrak{G}[V \setminus B])$ and $W_{\overline{\sigma}}(\mathfrak{G}) = W_\sigma(\mathfrak{G}[V \setminus B]) \cup B$ since $V \setminus B$ is a trap for player $\overline{\sigma}$ and player σ has no incentive to move to the attractor B (because it is winning for player $\overline{\sigma}$. The winning strategy for player σ is the winning strategy for the subgame $\mathfrak{G}[V \setminus B]$. The winning strategy for player $\overline{\sigma}$ is a combination of his winning strategy for the subgame $\mathfrak{G}[V \setminus B]$, the attractor strategy for B, and his winning strategy for the subgame $\mathfrak{G}[V \setminus A]$.

Given a finite parity game, the computational complexity of the algorithm is linear in the number of edges, polynomial in the number of vertices, and exponential in the number of colours [51]:

Proposition 2.10. Let $\mathfrak{G} = (\mathcal{A}, c)$ be a parity game with arena $\mathcal{A} = (V_0, V_1, E)$. Further more, let $n = |V|$ be the number of vertices, $m = |E|$ the number of edges, and $d = |\mathrm{img}(c)|$ the number of colours. The worst-case time complexity of the algorithm in Figure 2.9 is $O(m \times (n/d)^d)$.

Although a number of algorithms for solving parity games with better worst-case complexity have been proposed [51, 79], Zielonka's algorithm beats them in practice [44]. As an interesting side note, the parity game solving problem belongs to the UP $\cap$ co-UP complexity class[37] [50] and it is unknown whether a polynomial algorithm for solving it exists [54].

Generalised Parity Games

Generalised parity games were introduced in [28]. They refer to games with conjunctions or disjunctions of parity conditions. Here we consider the case that the goal of player 0 is a *conjunction* of parity conditions, i.e. he wants to ensure that for each colouring function, the colour occurring infinitely often is even. We can immediately see that the goal of player 1 is then a *disjunction* of parity conditions, i.e.

[37]Note that P $\subseteq$ UP $\cap$ co-UP $\subseteq$ NP $\cap$ co-NP.

```
function SolveParity($\mathfrak{G}$)
    if $V = \emptyset$ then
        return $(\emptyset, \emptyset)$                                ▷ Case 1.
    end if
    $p := \min\{c(v) \mid v \in V\}$
    $\sigma := p \bmod 2$
    $X := c^{-1}(p)$
    $A := \mathsf{attr}_\sigma(\mathcal{A}, X)$
    $(W_0, W_1) :=$ SolveParity($\mathfrak{G}[V \setminus A]$)
    if $W_{\overline{\sigma}} = \emptyset$ then                         ▷ Case 2.
        $W_\sigma = V$
    else                                                               ▷ Case 3.
        $B := \mathsf{attr}_{\overline{\sigma}}(\mathcal{A}, W_{\overline{\sigma}})$
        $(W_0, W_1) :=$ SolveParity($\mathfrak{G}[V \setminus B]$)
        $W_{\overline{\sigma}} := W_{\overline{\sigma}} \cup B$
    end if
    return $(W_0, W_1)$
end function
```

Figure 2.9: Algorithm for solving parity games [79]. It returns the winning regions (W_0, W_1) of a parity game $\mathfrak{G} = (\mathcal{A}, c)$ with arena $\mathcal{A} = (V_0, V_1, E)$ and colouring function $c : V \to \{0, \dots, k\}$ where $k \in \mathbb{N}$.

he wins if for at least one of the colouring functions, the colour occurring infinitely often is odd. Hence, the dual of a conjunctive parity game is a disjunctive parity game (and vice versa). This *asymmetry* between the two players is also reflected in the fact that player 1 (disjunction) always wins memoryless while player 0 (conjunction) requires finite memory in general.

The classical algorithm for solving generalised parity games is shown in Figure 2.10. Again, it returns the winning regions of the game. The winning strategies are as follows [28]:

- **Case 1.** The game has no vertices so there are no strategies.
- **Case 2.** Player 0 plays according to his winning strategy for the subgame $\mathfrak{G}[V \setminus A]$ when in W_0'. The winning strategy for player 1 depends on the current vertex v:
  - $v \in W_1'$: winning strategy for player 1 for the subgame $\mathfrak{G}[V \setminus A]$;
  - $v \in c_i^{-1}(p_i + 1) \cap G$: any move as long as the game stays within G;
  - $v \in \mathsf{attr}_1(A|_G, c_i^{-1}(p_i + 1) \cap G) \setminus (c_i^{-1}(p_i + 1) \cap G)$: (memoryless) attractor strategy;
  - $v \in W_1$: winning strategy for player 1 for the subgame $\mathfrak{G}[H]$;
  - $v \in A \setminus G$: (memoryless) attractor strategy.
- **Case 3.** All vertices are winning for player 0. In order to satisfy all conditions infinitely often, he needs to use the conjunct number i as memory (i.e. $M = \{1, \dots, n\}$). When his current memory state is i and the current vertex is v, player 0 uses the following strategy:
  - $v \in c_i^{-1}(p_i)$: choose any successor in G and update memory to $i + 1$;
  - $v \in \mathsf{attr}_0(\mathcal{A}, c_i^{-1}(p_i)) \setminus c_i^{-1}(p_i)$: attractor strategy (keep memory set to i);
  - otherwise, the following two cases apply. They refer to the value of W_0 and its attractor calculated on lines 14 and 15 in *some* iteration (not just the last one).
    - $v \in W_0$: winning strategy for player 0 for the subgame $\mathfrak{G}[H]$;
    - $v \in \mathsf{attr}_0(\mathcal{A}|_G, W_0) \setminus W_0$: (memoryless) attractor strategy.

Given a finite generalised parity game, the computational complexity of the algorithm is linear in the number of edges, polynomial in the number of vertices, and exponential in the number of colours of each colouring function [28]:

```
function SolveGenParity($\mathfrak{G}$)
    if $V = \emptyset$ then
        return $(\emptyset, \emptyset)$                                    ▷ Case 1.
    end if
    for $i := 1, \dots, k$ do
        $p_i := 2 \lfloor (\min \{c_i(v) \mid v \in V\}) / 2 \rfloor$        ▷ Ensure that $p_i$ is even.
    end for
    for $i := 1, \dots, k$ do
        if $p_i < k_i$ then
            $G := V \setminus \mathsf{attr}_0(\mathcal{A}, c_i^{-1}(p_i))$
            $H := G \setminus \mathsf{attr}_1(\mathcal{A}|_G, c_i^{-1}(p_i + 1) \cap G)$
            repeat
                $(W_0, W_1) :=$ SolveGenParity($\mathfrak{G}[H]$)
                $G := G \setminus \mathsf{attr}_0(\mathcal{A}|_G, W_0)$
                $H := G \setminus \mathsf{attr}_1(\mathcal{A}|_G, c_i^{-1}(p_i + 1) \cap G)$
            until $W_1 = \emptyset \vee W_1 = H$
            if $W_1 = H$ then
                $A := \mathsf{attr}_1(\mathcal{A}, G)$
                $(W_0', W_1') :=$ SolveGenParity($\mathfrak{G}[V \setminus A]$)
                return $(W_0', W_1' \cup A)$                               ▷ Case 2.
            end if
        end if
    end for
    return $(V, \emptyset)$                                                ▷ Case 3.
end function
```

Figure 2.10: Algorithm for solving generalised parity games [28]. It returns the winning regions (W_0, W_1) of a generalised parity game $\mathfrak{G} = (\mathcal{A}, \langle c_1, \dots, c_n \rangle)$ with arena $\mathcal{A} = (V_0, V_1, E)$ and colouring functions $c_i : V \to \{0, \dots, k_i\}$ where $k_i \in \mathbb{N}$.

Proposition 2.11. Let $\mathfrak{G} = (\mathcal{A}, \langle c_1, \ldots, c_k \rangle)$ be a generalised parity game with arena $\mathcal{A} = (V_0, V_1, E)$. Furthermore, let $n = |V|$ be the number of vertices, $m = |E|$ the number of edges, and $d_i = |\mathrm{img}(c_i)|$ the number of colours of colouring function c_i for all $i \in [1 \mathinner{..} k]$. The worst-case time complexity of the algorithm in Figure 2.10 is:

$$O\big(m \times n^d\big) \times \binom{\lceil d/2 \rceil}{\lceil d_1/2 \rceil, \ldots, \lceil d_k/2 \rceil} = O\big(m \times n^d\big) \times \frac{\lceil d/2 \rceil!}{\prod_{i=1}^{k} \lceil d_i/2 \rceil!}$$

where $d \triangleq \sum_{i=1}^{k} d_i$.

2.4.6 Summary

In this section, we gave a brief overview of concepts related to ω-automata and two-player games, which we will use to describe our model checking algorithm for SL[1G] in Chapter 5. We also explained how an arbitrary LTL formula can be converted to an equivalent deterministic parity automaton symbolically. Finally, we discussed how the winning regions and strategies of reachability games, parity games, and generalised parity games can be calculated.

2.5 Existing Tools

This section gives an overview of existing model checkers. We provide their key features, file formats, and supported specification languages. We also indicate whether they use BDD-based or SAT-based techniques (or both). The tools are presented in alphabetical order.

2.5.1 MCK

MCK is a model-checking tool for multi-agent systems developed at the School of Computer Science and Engineering at the University of New South Wales [7,14]. It uses mainly BDDs but also supports bounded model checking (BMC) and explicit state model checking (ESMC). The BDD packages supported by MCK are David Long's Binary Decision BDD package, BuDDy, and CUDD [1, 60, 85]. The basic logics for specifying properties supported by MCK are:

- LTL, CTL, CTL*;
- μ-calculus;
- Epistemic modalities including common knowledge;
- Fairness constraints.

In addition to the (traditional) observational semantics, MCK supports clock, asynchronous perfect recall, and synchronous perfect recall semantics. It also provides a GUI and counterexample generation. MCK is implemented in Haskell.

2.5.2 MCMAS

MCMAS is a *BDD-based* model-checker for the verification of multi-agent systems (MAS) developed at Imperial College London released under GNU General Public License (GPL) [5,8,63]. It uses the CUDD BDD package [85]. MAS descriptions are given in the form of programs in ISPL (Interpreted Systems Programming Language). MCMAS supports the following logics for specifying properties:

- CTL with fairness constraints;
- ATL with fairness constraints;
- Epistemic modalities;
- Deontic modalities expressing correctness of agents' behaviour[38] [66].

[38]The syntax is extended with the deontic modality $\mathsf{O}_i\, \varphi$. Informally, $\mathsf{O}_i\, \varphi$ is true iff φ is true in all states in which agent $i \in \mathit{Agt}$ is functioning correctly.

MCMAS also supports generating counterexample traces and provides a GUI in the form of an Eclipse plug-in, whose functionalities include ISPL program editing with dynamic syntax checking, interactive execution mode, and counterexample/witness display [63]. MCMAS is implemented in C++.

As part of this project, we implemented two extensions for MCMAS which add support for Slk and Sl[1g] model checking (see Sections 6.2 and 6.3). Therefore, MCMAS functionality, usage, and architecture is described in much more detail in Section 6.1. Nevertheless, we decided to include it in this section as well so that it could be easily compared with other existing tools.

2.5.3 Mocha

Mocha is a BDD-based model checker developed jointly at the University of California at Berkeley, the University of Pennsylvania, and the State University of New York at Stony Brook [17]. It differs from conventional model checkers in that its main purpose is to facilitate the development of new verification techniques. Systems are modelled using reactive modules, which represent their synchronous, asynchronous, and real-time components. Mocha supports the following formalisms for specifying properties [16]:

- Atl;
- Invariants (propositional formulas which are intended to be true in all reachable states[39]);
- Abstract modules, which should be implemented by the system module (refinement).

Mocha provides a GUI for interactive simulation. It is currently available in two versions, cMocha and jMocha, implemented in C and Java respectively. The latter currently does not support Atl model checking.

2.5.4 NuSMV

NuSMV is an open-source tool for model checking published under GNU Lesser General Public License 2.1 (LGPL) which supports both BDD-based and SAT-based model checking [6, 10, 30]. NuSMV is a reimplementation and extension of SMV [69]. It uses the CUDD BDD package and supports the Z-Chaff and MiniSat SAT-solvers [3, 36, 85].

NuSMV processes files in the SMV format, which it first translates to a finite state machine (FSM) and then performs either BDD-based model checking, or bounded model checking (BMC) [32]. Figure 2.11 shows an SMV file and the corresponding finite state machine. The tool supports the following logics for specifying properties [31]:

- Ltl using either the tableau method [34], in which it is automatically converted to a Ctl formula and verified on a tableau with fairness constraints (BDD-based), or using BMC (SAT-based);
- Ctl with fairness constraints (BDD-based);
- RTCtl (Real Time Ctl [40]) which augments Ctl with real-time constraints[40] (BDD-based);
- A subset of Psl (Property Specification Language [15]).

NuSMV also supports generating counterexample traces and provides an interactive shell [31]. It is implemented in C.

2.5.5 PRISM

PRISM is a probabilistic model checker released under the GNU General Public License [5, 11, 57]. It uses various techniques including BDDs, MTBDDs[41], discrete-event simulation, quantitative abstraction refinement, and symmetry reduction. PRISM supports a wide range of models:

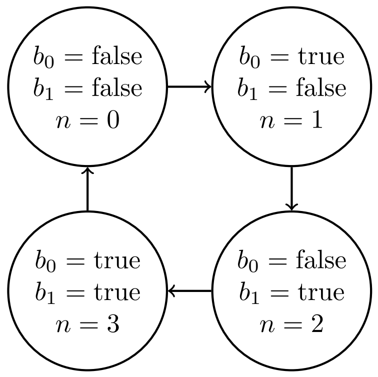
[39]Note that checking an invariant φ is equivalent to checking the Ctl formula $\mathsf{AG}\,\varphi$.

[40]For example, the RTCtl formula $\mathsf{AG}\left(\alpha \rightarrow \mathsf{AF}^{0..4}\,\beta\right)$ expresses that whenever α is true, β is true in 0 to 4 steps [31].

[41]Multi-terminal binary decision diagrams (MTBDDs) are a generalisation of BDDs in which terminal nodes are labelled with arbitrary real values (not just 0 and 1) [56, Section 4].

```
MODULE main
  VAR
  b0 : boolean;
  b1 : boolean;
  n : {0, 1, 2, 3};
ASSIGN
  init(b0) := FALSE;
  init(b1) := FALSE;
  init(n) := 0;
  next(b0) := !b0;
  next(b1) := b1 xor (b0 & !next(b0));
  next(n) := (n + 1) mod 4;
LTLSPEC
  G (b0 xor X b0);
CTLSPEC
  AG (b0 <-> (n mod 2) = 1);
CTLSPEC
  AG (b1 <-> (n / 2) = 1);
```

(a) SMV file.



(b) Finite state machine.

Figure 2.11: Sample SMV file (with one Ltl property and two Ctl properties) and the corresponding finite state machine.

- Discrete deterministic – Discrete-time Markov chains (DTMCs);
- Discrete non-deterministic – Markov decision processes (MDPs) and probabilistic automata (PAs);
- Continuous deterministic – Continuous-time Markov chains (CTMCs);
- Continuous non-deterministic – Probabilistic timed automata (PTAs) and Priced probabilistic timed automata (PPTAs).

and the extensions of these models with costs and rewards [57]. PRISM's property specification language[42] subsumes the following logics:

- PCtl (Probabilistic computation tree logic [22]) and PCTL* which augment Ctl and Ctl* with probability bounds[43];
- Csl (Continuous stochastic logic [20]) for continuous systems inspired by Ctl[44];
- Ltl;
- Most of Ctl.

PRISM also provides optimal adversary/strategy generation for nondeterministic models and a GUI. It is implemented in a mixture of Java and C++.

2.5.6 VerICS

VerICS is a *SAT-based* model-checking tool for multi-agent systems developed at the Institute of Computer Science, Polish Academy of Sciences [52]. It uses bounded model checking (BMC) and unbounded

[42]For example the property "`P<0.1 [ F<=100 num_errors > 5 ]`" expresses that "the probability that more than 5 errors occur within the first 100 time units is less than 0.1" [11].

[43]The syntax is extended with the probabilistic operator P. Intuitively, $\mathsf{P}_{\geq p}\varphi$ ($\mathsf{P}_{\leq p}\varphi$) means that φ holds with probability at least (at most) p.

[44]Informally, Csl extends propositional logic with formulas of the form $\mathsf{P}_{>p}\,\phi$ where $\phi = \big(\varphi_1 \,\mathsf{U}_{[a_1,b_1]}\, \varphi_2 \,\mathsf{U}_{[a_2,b_2]} \cdots \varphi_n\big)$. ϕ is true on a path π iff there exist real numbers $t_1, \ldots t_{n-1}$ such that for each integer $i \in [1 \mathinner{..} n-1]$ we have $a_i \leq t_i \leq b_i$ and $\forall t' \in [t_{i-1}, t_i) \,.\, \pi(t') \models_{\text{CSL}} \varphi_i$ where $t_0 = 0$.

model checking (UMC) to verify temporal, epistemic, and deontic properties on timed automata and timed Petri nets. BMC uses the MiniSat or RSat SAT-solver and UMC uses a modified version of the Z-Chaff SAT-solver [3, 12, 36]. The logics supported by VerICS include:

- CTL_pK (CTL with past and knowledge operators) using UMC;
- ECTLKD (existential fragment of CTL with knowledge and deontic operators) using BMC;
- TECTLK (existential fragment of timed CTL with knowledge operators) using BMC.

VerICS provides a GUI for modelling timed automata and timed Petri nets. It is implemented in Java.

2.6 Summary

In this chapter, we provided the background theory that the rest of this report is based on. We explained that formal verification techniques typically comprise three components, a framework for modelling systems, a specification language, and a verification method. *Model checking*, which is a model-based verification method, refers to the process of determining whether a property P encoded as a formula ϕ_P holds in a system S represented by a model $\mathcal{M}_S$, i.e. $\mathcal{M}_S \models \phi_P$. We then discussed each of the three components of verification using model checking.

We first compared the most common frameworks for modelling systems and explained why interpreted systems are best suited for modelling multi-agent systems. As a second step, we described several logics of increasing expressiveness for specifying temporal properties of a system and demonstrated the power of SL. We also gave a brief overview of epistemic modalities, which express agents' knowledge. We concluded our tour of specification languages by pointing out the close relationship between expressiveness and model checking complexity.

We then discussed verification methods. We introduced BDDs, which are commonly used for symbolic model checking because of their ability to handle large state spaces efficiently, and explained how they can represent sets and relations. A large portion of the chapter was devoted to the theory of ω-automata and two-player infinite games, which are central to the model checking algorithms of many temporal logics including LTL.

Finally, we gave an overview of state of the art model checkers and compared their functionality and architecture. Before moving on to the next chapter, we would like to point out that none of them supports either ATL* or SL.

Chapter 3

Fragment Selection

Our aim is to build a model checker for SL. As explained in Subsection 2.2.5, SL is very expressive. Unlike ATL*, which subsumes all other logics presented in Section 2.2, SL can express complex game-theoretic concepts like Nash equilibria. However, as we have already pointed out, such power comes at a cost. In fact, there are several problems with the original SL defined on perfect recall semantics:

1. **Complexity.** The model checking of SL is NONELEMENTARY with respect to the size of the formula. More precisely, model checking an SL sentence φ with alternation number k is k-EXPSPACE-HARD and $(k+1)$-EXPTIME [72]. This is far worse than the 2EXPTIME-COMPLETE model checking complexity of ATL* [18]. This suggests that, given the computational power of today's computers, any model checking algorithm would be able to verify only very simple SL formulas, rendering SL expressiveness in practice very limited. Nevertheless, it is worth pointing out the the complexity is still P-COMPLETE with respect to the size of the model. Hence, it should be possible to check such simple SL formulas against very large models.

2. **Non-behavioural strategies.** SL admits *non-behavioural strategies* where an agent's action might depend on another agent's action in another counterfactual game [75]. Given that an agent clearly cannot predict another agent's behaviour, such strategies are *not synthesisable* in practice. This is a major drawback since we want to know not only *whether* a formula is true or false, but also *why* it is the case, i.e. what strategies should the agents employ in order to either enforce it, or violate it.

3. **Undecidability under incomplete information.** SL is undecidable under perfect recall and incomplete information[1]. This means that we cannot augment SL with epistemic modalities, which require incomplete information, unless agents have imperfect recall. Moreover, without incomplete information, the synthesised strategies might not be *uniform*, i.e. an agent might have to perform different actions in two indistinguishable local states. Such strategies are of limited use in models where agents do not have complete knowledge of the system (e.g. interpreted systems discussed in Subsection 2.1.3).

To sum up, model checking the original SL (with complete information) would probably have very low performance and limited application. Therefore, we decided to consider *fragments* of SL, which do not suffer from these limitations. Given the undecidability of SL under perfect recall and incomplete information, we had basically two options[2]:

1. SL with *imperfect recall* (and incomplete information) and

2. SL with *complete information* (and perfect recall).

We considered both options and identified two fragments of SL, namely *Epistemic Strategy Logic* (SLK) and *One-Goal Strategy Logic* (SL[1G]), for which we designed model checking algorithms presented in Chapters 4 and 5.

[1]A proof of undecidability of ATL under incomplete information and perfect recall was presented in [35]. Since SL (strictly) subsumes ATL, our claim follows.

[2]SL with *complete* information and imperfect recall is a subset of SL with *incomplete* information and imperfect recall. Therefore, we do not discuss this option separately.

3.1 Imperfect Recall

The first way to tackle the issues outlined at the beginning of this chapter is to use imperfect recall semantics with incomplete information, under which agents have no memory of the past and do not have complete knowledge of the global state. Hence, their actions depend solely on their current local states, i.e. they use memoryless strategies $f_i : L_{i\mathrm{E}} \to Act_i$. This set up, also referred to as *memoryless semantics*, addresses most of the main problems associated with full SL:

1. The model checking complexity with respect to the size of the formula is PSPACE (see Theorem 4.1). This is a *massive improvement* compared to SL with perfect recall. However, the model checking complexity with respect to the size of the model is now also PSPACE.

2. SL with imperfect recall still admits non-behavioural strategies. Moreover, it appears that under incomplete information, behavioural semantics cannot be obtained even by restricting the syntax of SL as we will do in Section 3.2.

3. Incomplete information is decidable under imperfect recall. This should not come as a surprise given that the number of local states, actions, and hence memoryless strategies is finite. Intuitively, we could explicitly enumerate all possible strategies each time a quantifier is encountered.

Although SL with imperfect recall admits non-behavioural strategies, we believe that it has great theoretical value as it supports reasoning about game-theoretic concepts under incomplete information. Furthermore, we can increase its expressiveness by adding epistemic modalities (see Subsection 2.2.6), which turn out to have no impact on either complexity or decidability. We refer to this new variant of SL with imperfect recall, incomplete information, and epistemic modalities as *Epistemic Strategy Logic*, or Strategy Logic with Knowledge (SLK). We formally define it and provide a model checking algorithm for it in Chapter 4.

3.2 Complete Information

Another approach for addressing the undecidability problem is to use complete information semantics, under which agents have complete knowledge of the whole model. Their actions thus depend on the whole history of global states, i.e. they use strategies[3] $f_i : Trk \to Act_i$. This set up was used by the original SL, introduced in [76], and it still suffers from the first two problems, namely NONELEMENTARYSPACE-HARD model checking complexity and non-behavioural strategies.

To address this issue, several *syntactic fragments* of SL have been proposed [72, 75]. The fragments use LTL syntax (see Definition 2.7) augmented with the quantification rule $\wp\psi$ where $\wp$ is a quantification prefix[4] for all free variables in ψ, ψ is an SL formula consisting of agent-closed *goals* of the form $\flat\varphi$, $\flat$ is a binding prefix, and φ is an LTL formula. The syntactic fragments differ in the combinations of goals they permit. For example, the following is a Boolean-Goal Strategy Logic formula:

$$\underbrace{[\![x]\!]\langle\!\langle y\rangle\!\rangle[\![z]\!]}_{\text{quant. prefix }\wp}\,[\underbrace{\overbrace{(a,x)(b,y)(c,z)}^{\text{binding prefix }\flat_1}\mathsf{X}\,p}_{\text{goal }\flat_1\varphi_1} \vee \underbrace{(a,y)(b,z)(c,x)\overbrace{(\mathsf{F}\,\mathsf{G}\,q \to \mathsf{F}\,\mathsf{G}\,r)}^{\text{LTL formula }\varphi_2}}_{\text{goal }\flat_2\varphi_2}]$$

Informally, the fragments differ in which operators can occur "between" a quantification prefix $\wp$ and goals $\flat\varphi$. The following syntactic fragments have been proposed in [72, 75] (k is alternation number of the formula):

- **Nested-Goal Strategy Logic** (SL[NG]).

Syntax:	$\psi ::= \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathsf{X}\,\psi \mid \psi\,\mathsf{U}\,\psi \mid \flat\psi \mid \varphi$ (any LTL operator or bind. prefix)
Complexity:	NONELEMENTARY (k-EXPSPACE-HARD lower b., $(k+1)$-EXPTIME upper b.)
Semantics:	non-behavioural
Satisfiability:	undecidable

[3] Recall that $Trk \subseteq G^+$ is the set of all possible *tracks* (non-empty finite sequences of global states) in the underlying interpreted system (see Definition 2.6).

[4] We define the quantification and binding prefixes formally when we introduce SL[1G] syntax (see Definition 5.1).

- **Boolean-Goal Strategy Logic** (SL[BG]).

Syntax:	$\psi ::= \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \flat\varphi$ (any Boolean operator)
Complexity:	open question (2EXPTIME-HARD lower b., $(k+1)$-EXPTIME upper b.)
Semantics:	non-behavioural
Satisfiability:	undecidable

- **Conjunctive-Goal Strategy Logic** (SL[CG]).

Syntax:	$\psi ::= \psi \wedge \psi \mid \flat\varphi$ (conjunction only)
Complexity:	2EXPTIME-COMPLETE
Semantics:	behavioural
Satisfiability:	open problem

- **Disjunctive-Goal Strategy Logic** (SL[DG]).

Syntax:	$\psi ::= \psi \vee \psi \mid \flat\varphi$ (disjunction only)
Complexity:	2EXPTIME-COMPLETE
Semantics:	behavioural
Satisfiability:	open problem

- **One-Goal Strategy Logic** (SL[1G]).

Syntax:	$\psi ::= \flat\varphi$ (quantification coupled with binding)
Complexity:	2EXPTIME-COMPLETE
Semantics:	behavioural
Satisfiability:	decidable

Note that SL[DG] and SL[CG] are duals of each other. Therefore, we will refer to them jointly[5] as SL[DG/CG]. The complete expressiveness chain of all syntactic fragments, the original SL, and ATL* together with their model checking complexities is [72, 75]:

$$\underbrace{\text{ATL}^* < \text{SL[1G]} < \text{SL[DG/CG]}}_{\text{2EXPTIME-COMPLETE}} < \underbrace{\text{SL[BG]}}_{?} \leq \underbrace{\text{SL[NG]} \leq \text{SL}}_{\substack{\text{NONELEMENTARY-}\\ \text{SPACE-HARD}}}$$

Clearly, there are two candidates for our purposes: SL[1G] and SL[DG/CG]. Both of them have a much lower model checking complexity than SL and admit only behavioural strategies. In addition, they have the same model checking complexity as ATL* while being strictly more expressive. Although there is a difference between SL[1G] and SL[DG/CG], namely that the satisfiability of SL[1G] is known to be decidable but still remains an open problem for SL[DG/CG], it is not relevant for our purposes and thus not a strong criterion for fragment selection.

Given the amount of time for the project and the theoretical complexity of model checking and strategy synthesis under perfect recall, we decided to focus on SL[1G]. We provide a novel model checking algorithm for SL[1G] with perfect recall and complete information in Chapter 5. It is perfectly possible that only minor modifications of the algorithm would be necessary in order to support SL[DG/CG] as well.

3.3 Toy Model

Some of the model checking concepts presented in Chapters 4 and 5 are very abstract and might be difficult to grasp on first reading. In order to make them easier to understand for the reader, we will demonstrate them on a very simple toy model, a game of *Rock-Paper-Scissors*. It is a traditional two-player game, where both players simultaneously show a gesture (using their hands) representing one of the three objects. This pattern is repeated, until each of the players shows a different gesture. Once their gestures differ, the winner is determined according to the following rules: rock beats scissors, scissors beat paper, and paper beats rock.

The simplicity of this game will allow us to provide detailed examples for almost all concepts and procedures introduced in this thesis. However, it is very important to stress that *this toy model is in*

[5]This does *not* mean that both disjunctions and conjunctions can be used within one prefix. Such a fragment would be equivalent to SL[BG] since $\neg\flat\varphi \equiv \flat\neg\varphi$ (i.e. we could propagate all negations using De Morgan's laws into the goals).

no way representative of the expressive power of SL *or its fragments.* Unfortunately, if we used a more realistic example, some of the constructions could not be fully presented in this thesis as they would have hundreds or even thousands of states. In fact, as we will see in Chapter 5, even this simple model (with only 3 states) generates a parity game with more than 50 states when model checking the SL[1G] formula $\mathsf{X}\, p$ (see Figure 5.2). More compelling examples, including specifications expressing Nash equilibria and agents' knowledge, will be presented in Section 6.4.

3.3.1 Formal Definition

Let us now define the model more formally as an interpreted system $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ (see Definition 2.5). To model the two-player game, we use three agents $Agt = \{1, 2, \mathrm{E}\}$, where E represents the environment. The internal states L_i and actions Act_i of the agents are as follows:

$$\begin{aligned} L_1 &\triangleq \{s\} & Act_1 &\triangleq \{\mathrm{r}, \mathrm{p}, \mathrm{s}, \mathrm{i}\} \\ L_2 &\triangleq \{s\} & Act_2 &\triangleq \{\mathrm{r}, \mathrm{p}, \mathrm{s}, \mathrm{i}\} \\ L_\mathrm{E} &\triangleq \{s_\mathrm{g}, s_1, s_2\} & Act_\mathrm{E} &\triangleq \{\mathrm{i}\} \end{aligned}$$

Both agents have only one internal state s, i.e. all state information is kept in the environment (and hence agents have complete information). Conversely, the environment can only perform the idle action i. The environment's states s_g, s_1, and s_2 stand for "game", "player 1 victory", and "player 2 victory" respectively. There are 3 global states[6] $G \triangleq L_1 \times L_2 \times L_\mathrm{E} = \{(s, s, s_\mathrm{g}), (s, s, s_1), (s, s, s_2)\}$, which we will denote g_g, g_1, and g_2 for conciseness. The set of initial states is $I \triangleq \{g_\mathrm{g}\}$. The protocols of the agents are as follows:

$$\begin{aligned} P_1(s, s_\mathrm{g}) &\triangleq \{\mathrm{r}, \mathrm{p}, \mathrm{s}\} & P_1(s, s_1) &\triangleq \{\mathrm{i}\} & P_1(s, s_2) &\triangleq \{\mathrm{i}\} \\ P_2(s, s_\mathrm{g}) &\triangleq \{\mathrm{r}, \mathrm{p}, \mathrm{s}\} & P_2(s, s_2) &\triangleq \{\mathrm{i}\} & P_2(s, s_2) &\triangleq \{\mathrm{i}\} \\ P_\mathrm{E}(s_\mathrm{g}) &\triangleq \{\mathrm{i}\} & P_\mathrm{E}(s_1) &\triangleq \{\mathrm{i}\} & P_\mathrm{E}(s_2) &\triangleq \{\mathrm{i}\} \end{aligned}$$

Furthermore, there are 16 joint actions, only 10 of which are possible:

$$\begin{aligned} Act \triangleq Act_1 \times Act_2 \times Act_\mathrm{E} = \{\, & (\mathrm{r}, \mathrm{r}, \mathrm{i}), (\mathrm{r}, \mathrm{p}, \mathrm{i}), (\mathrm{r}, \mathrm{s}, \mathrm{i}), \cancel{(\mathrm{r}, \mathrm{i}, \mathrm{i})}, (\mathrm{p}, \mathrm{r}, \mathrm{i}), (\mathrm{p}, \mathrm{p}, \mathrm{i}), (\mathrm{p}, \mathrm{s}, \mathrm{i}), \cancel{(\mathrm{p}, \mathrm{i}, \mathrm{i})}, \\ & (\mathrm{s}, \mathrm{r}, \mathrm{i}), (\mathrm{s}, \mathrm{p}, \mathrm{i}), (\mathrm{s}, \mathrm{s}, \mathrm{i}), \cancel{(\mathrm{s}, \mathrm{i}, \mathrm{i})}, \cancel{(\mathrm{i}, \mathrm{r}, \mathrm{i})}, \cancel{(\mathrm{i}, \mathrm{p}, \mathrm{i})}, \cancel{(\mathrm{i}, \mathrm{s}, \mathrm{i})}, (\mathrm{i}, \mathrm{i}, \mathrm{i}) \,\} \end{aligned}$$

To avoid cluttered notation, we will denote them $\mathrm{rr}, \mathrm{rp}, \ldots, \mathrm{ii}$ (using only the actions of the two original players). The transition functions of agents 1 and 2 are defined as $t_i(s, a) \triangleq s$ for all joint actions $a \in Act$. The environment evolution function is defined as follows:

$$\begin{aligned} t_\mathrm{E}(s_\mathrm{g}, \mathrm{rr}) &\triangleq s_\mathrm{g} & t_\mathrm{E}(s_\mathrm{g}, \mathrm{pr}) &\triangleq s_1 & t_\mathrm{E}(s_\mathrm{g}, \mathrm{sr}) &\triangleq s_2 & t_\mathrm{E}(s_1, \mathrm{ii}) &\triangleq s_1 \\ t_\mathrm{E}(s_\mathrm{g}, \mathrm{rp}) &\triangleq s_2 & t_\mathrm{E}(s_\mathrm{g}, \mathrm{pp}) &\triangleq s_\mathrm{g} & t_\mathrm{E}(s_\mathrm{g}, \mathrm{sp}) &\triangleq s_1 & t_\mathrm{E}(s_2, \mathrm{ii}) &\triangleq s_2 \\ t_\mathrm{E}(s_\mathrm{g}, \mathrm{rs}) &\triangleq s_1 & t_\mathrm{E}(s_\mathrm{g}, \mathrm{ps}) &\triangleq s_2 & t_\mathrm{E}(s_\mathrm{g}, \mathrm{ss}) &\triangleq s_\mathrm{g} \end{aligned}$$

Finally, we introduce two propositional variables p_1 and p_2 with $h(p_1) \triangleq \{g_1\}$ and $h(p_2) \triangleq \{g_2\}$. The complete interpreted system is shown in Figure 3.1.

3.3.2 Symbolic Implementation

Here we describe how the interpreted system for the toy problem formalised in Subsection 3.3.1 can be encoded symbolically (see Subsection 2.3.2 for a brief overview of symbolic model checking). The approach we describe here is used by the MCMAS model checker (see Subsection 2.5.2).

We start by describing how to encode internal states and actions. A set of internal states L_i can be encoded using $\log_2 \lceil |L_i| \rceil$ variables. Since both L_1 and L_2 contain only one element, we do not need any

[6]In Definition 2.5, we use G to denote the set of *reachable* states, which is a subset of the set of global states, i.e. $G \subseteq L_1 \times \cdots \times L_n \times L_\mathrm{E}$. However, it turns out that the set of reachable states is equal to the set of global states in this model.

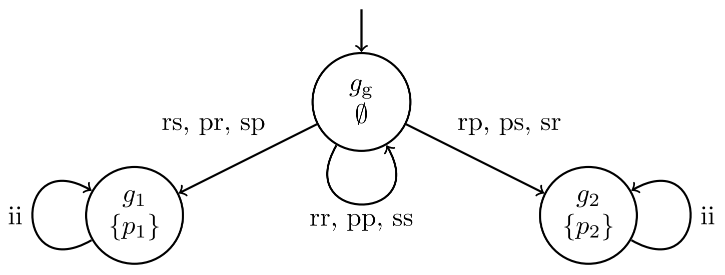


Figure 3.1: Representation of the toy model as an interpreted system.

variables to encode them. L_E can be represented using $\log_2 \lceil |L_E| \rceil = \log_2 \lceil 3 \rceil = 2$ Boolean variables. Thus, the global states G can be encoded using a Boolean vector $\overline{v} = [v_0, v_1]$:

$$g_g(\overline{v}) = \neg v_0 \wedge \neg v_1 \qquad g_1(\overline{v}) = v_0 \wedge \neg v_1 \qquad g_2(\overline{v}) = \neg v_0 \wedge v_1$$

In addition, we need 2 Boolean variables v_0' and v_1' to represent the next state in the transition relation. Since g_g is the only initial state, we have $I(\overline{v}) = \neg v_0 \wedge \neg v_1$. Actions are encoded in a very similar manner. Hence, we need no variables to represent the action of the environment and each of the two original players requires 2 Boolean variables to represent their 4 possible actions. A joint action can thus be encoded using a Boolean vector $\overline{w} = [w_0, w_1, w_2, w_3]$. For example, the encoding of the joint action sp is:

$$\mathrm{sp}(\overline{w}) = \underbrace{\neg w_0 \wedge w_1}_{\mathrm{s}} \wedge \underbrace{w_2 \wedge \neg w_3}_{\mathrm{p}}$$

In general, a protocol P_i and an evolution function t_i are encoded as follows [65]:

$$P_i(\overline{v}, \overline{w}) = \bigvee_{l_{i\mathrm{E}} \in L_{i\mathrm{E}}} \left[l_i(\overline{v}_{i\mathrm{E}}) \wedge \bigvee_{a_i \in Act_i} a_i(\overline{w}_i) \right]$$
$$t_i(\overline{v}, \overline{w}, \overline{v'}) = \bigvee_{l_{i\mathrm{E}} \in L_{i\mathrm{E}}} \bigvee_{a \in Act} \left[l_{i\mathrm{E}}(\overline{v}_i) \wedge a(\overline{w}) \wedge t_i(l_{i\mathrm{E}}, a)(\overline{v'}) \right]$$

For example, the encoding of the protocol P_1 is:

$$P_1(\overline{v},\overline{w}) = \underbrace{\overbrace{\neg v_0 \wedge \neg v_1}^{s_g} \wedge \big(\overbrace{\neg w_0 \wedge \neg w_1}^{\mathrm{r}} \vee \overbrace{w_0 \wedge \neg w_1}^{\mathrm{p}} \vee \overbrace{\neg w_0 \wedge w_1}^{\mathrm{s}}\big)}_{P_1(s,s_g)=\{\mathrm{r,p,s}\}} \vee \underbrace{\overbrace{v_0 \wedge \neg v_1}^{s_1} \wedge \overbrace{w_0 \wedge w_1}^{\mathrm{i}}}_{P_1(s,s_1)=\{\mathrm{i}\}} \vee \underbrace{\overbrace{v_0 \wedge \neg v_1}^{s_2} \wedge \overbrace{w_0 \wedge w_1}^{\mathrm{i}}}_{P_1(s,s_2)=\{\mathrm{i}\}}$$

If we define the global protocol and evolution function as $P(\overline{v}, \overline{w}) = \bigwedge_{i \in Agt} P_i(\overline{v}, \overline{w})$ and $t(\overline{v}, \overline{w}, \overline{v'}) = \bigwedge_{i \in Agt} t_i(\overline{v}, \overline{w}, \overline{v'})$ respectively, we can derive an expression for the binary transition relation:

$$R(\overline{v}, \overline{v'}) = \exists \overline{w}. P(\overline{v}, \overline{w}) \wedge t(\overline{v}, \overline{w}, \overline{v'})$$

Finally, we can calculate the fixpoint for reachable states[7] $G = \mathrm{lfp}_Q \left[I \cup \mathsf{suc}_\exists(Q) \right]$ symbolically as:

$$G(\overline{v}) = \mathrm{lfp}_\Theta \big[I(\overline{v}) \vee \underbrace{\exists \overline{v'}. \left(\Theta' \wedge R(\overline{v'}, \overline{v}) \right)}_{\mathsf{suc}_\exists(Q)} \big]$$

where Θ' is a Boolean formula equal to Θ with variables in $\overline{v}$ and $\overline{v'}$ swapped.

This completes the symbolic implementation of the toy model. ISPL code[8] for the model is provided in Appendix A.

[7]$\mathsf{suc}_\exists$ is the opposite of $\mathsf{pre}_\exists$ (see Definition 2.11). It calculates the set of *successors* of a set of global states $X \subseteq G$ in an interpreted system $\mathcal{I}$: $\mathsf{suc}_\exists(X) \triangleq \{g \in G \mid \exists g' \in X \exists a \in Act.\, t(g', a) = g\}$.

[8]As explained in Subsection 2.5.2, ISPL is a language for modelling multi-agent systems supported by MCMAS. The ISPL syntax is described in Subsection 6.1.2.

3.4 Summary

In this short chapter, we discussed the problems associated with model checking full SL, namely NONELEMENTARYSPACE-HARD model checking complexity, non-behavioural strategies, and undecidability under incomplete information. In order to tackle these issues, we identified two *fragments* of SL, Epistemic Strategy Logic (SLK) and One-Goal Strategy Logic (SL[1G]), which we will further investigate in Chapters 4 and 5 respectively.

Furthermore, we provided a very simple *toy model* of the game of Rock-Paper-Scissors, which we will use throughout the rest of this report to demonstrate new concepts. We encoded it as an interpreted system and explained how it can be implemented symbolically.

Chapter 4

Epistemic Strategy Logic

In this chapter, we introduce *Epistemic Strategy Logic*, or Strategy Logic with Knowledge (SLK), a modification of SL (see Subsection 2.2.5) which is defined on imperfect recall semantics with incomplete information (i.e. agents have no memory of the past and do not have a complete knowledge of the global state of the system). We show that the corresponding model checking problem belongs to the PSPACE complexity class. We also provide an efficient model checking algorithm for it and prove its correctness. Both the logic and the algorithm constitute original material developed as part of this individual project. This chapter is split into two parts: Section 4.1 introduces the logic and Section 4.2 describes the model checking algorithm.

4.1 Logic

SLK is a logic which combines the ability of SL to express game-theoretic concepts with the epistemic framework for describing agents' knowledge. It is a unique blend of three well-established formalisms: *(i)* LTL operators X, F, G, and U, *(ii)* SL strategy quantifiers $\langle\!\langle x \rangle\!\rangle$, $[\![x]\!]$ and agent binding (a, x), and *(iii)* epistemic modalities K_i, E_A, D_A, and C_A. This structure allows us to reason about the temporal, behavioural, and epistemic aspects of a model separately. As explained in Section 3.1, the logic is defined on imperfect recall semantics with incomplete information in order for the model checking problem to be decidable.

This section provides formal definitions of SLK syntax, semantics, and other related concepts. The definitions will provide us with a solid foundation for the development of the model checking algorithm presented in the next section. We will also discuss several examples and demonstrate some limitations of SLK.

4.1.1 Syntax

The syntax of SLK extends the syntax of SL (Definition 2.19) with epistemic modalities representing individual knowledge K_i, group knowledge E_A, distributed knowledge D_A, and common knowledge C_A, where $i \in Agt$ is an agent and $A \subseteq Agt$ a set of agents (see Subsection 2.2.6 for a brief introduction to epistemic modalities).

Definition 4.1 (SLK Syntax)**.** SLK *formulas* are built inductively using the following grammar, where $p \in AP$ is an atomic proposition, $x \in Var$ a variable, $i \in Agt$ an agent, and $A \subseteq Agt$ a set of agents:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid$$
$$\langle\!\langle x \rangle\!\rangle\varphi \mid [\![x]\!]\varphi \mid (i, x)\varphi \mid \mathsf{K}_i\,\varphi \mid \mathsf{E}_A\,\varphi \mid \mathsf{D}_A\,\varphi \mid \mathsf{C}_A\,\varphi$$

where epistemic operators are applied to sentences only, i.e. $\mathsf{free}(\varphi) = \emptyset$ in $\mathsf{K}_i\,\varphi$, $\mathsf{E}_i\,\varphi$, $\mathsf{D}_A\,\varphi$, and $\mathsf{C}_A\,\varphi$. *SLK* denotes the infinite set of formulas generated by the above rules.

We use the set of free agents/variables as in Definition 2.20 with an extra rule: $\mathsf{free}(\mathsf{K}_i\,\varphi) \triangleq \mathsf{free}(\mathsf{E}_A\,\varphi) \triangleq \mathsf{free}(\mathsf{D}_A\,\varphi) \triangleq \mathsf{free}(\mathsf{C}_A\,\varphi) \triangleq \emptyset$, for all $i \in Agt$ and $A \subseteq Agt$. The reason for this is that φ

must be a sentence according to the definition above. For example, given the set of agents $Agt = \{\mathrm{E}, a, b\}$ and an agent $i \in Agt$, $[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!](\mathrm{E}, e)(a, x)(b, y)\mathsf{K}_i\mathsf{X}\, p$ and $[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!]\mathsf{K}_i(\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, p$ are not well-formed SLK formulas since $\mathsf{free}(\mathsf{X}\, p) = \{\mathrm{E}, a, b\}$ and $\mathsf{free}((\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, p) = \{e, x, y\}$. On the other hand, $[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!](\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, \mathsf{K}_i\, p$ and $\mathsf{K}_i[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!](\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, p$ are well-formed because both p and $[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!](\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, p$ are sentences, i.e. $\mathsf{free}(p) = \mathsf{free}([\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!](\mathrm{E}, e)(a, x)(b, y)\mathsf{X}\, p) = \emptyset$.

Observe that the syntax of SLK strictly subsumes all logics in the ATL* hierarchy with imperfect recall and incomplete information including CTLK and ATLK. Moreover, it allows us to express properties which could not be expressed by any of the previous logics (including the original SL). For example, the SLK formula:

$$\mathsf{E}_{\{a,b\}}\ [\![e]\!][\![x]\!](\mathrm{E}, e)(a, x)(b, x)[\mathsf{F}\,\mathsf{G}\, p \wedge \mathsf{F}\,\mathsf{G}\,\neg p]$$

expresses that *"the agents a and b both know that if they use the same strategy, p will be infinitely often true and infinitely often false"*. This formula demonstrates that the large expressive power of SLK comes from the combination of temporal, strategic, and epistemic modalities.

4.1.2 Basic Concepts

Before defining the semantics of SLK, we have to introduce some basic concepts similar to the ones introduced in Subsection 2.2.5. Essentially, we have to cater for the fact that agents now have incomplete information and imperfect recall.

Let us consider agents' strategies. Since agents now have imperfect recall (no memory), their next actions depend on their current state only. Furthermore, in order for a strategy to be executable by an agent with incomplete information, it must assign the same action to all global states that the agent cannot distinguish, i.e. when $g_1, g_2 \in G$ are two global states such that $g_1 \sim_i g_2$, it must be the case that the strategy of agent $i \in Agt$ satisfies $f_i(g_1) = f_i(g_2)$. This property is sometimes referred to as Γ-*uniformity* [65]. Since SLK strategies can be shared among multiple agents, it must apply to all of them. Hence, an SLK strategy must *(i)* comply with the protocols of all agents that share it and *(ii)* map any two global states that cannot be distinguished by any of the agents to the same action. A formal definition follows.

Definition 4.2 (Uniform Shared Memoryless Strategies)**.** A *uniform shared memoryless strategy* of a non-empty set of agents $A \subseteq Agt$ in an interpreted system $\mathcal{I}$ is a function $f : G \to Act_A$ such that the following two conditions hold:

1. **Shared.** For all global states $g \in G$ and agents $i \in A$, $f(g) \in P_i(l_{i\mathrm{E}}(g))$, i.e. agents comply with their protocols.

2. **Uniform.** For all global states $g_1, g_2 \in G$ and agents agents $i \in A$, if $g_1 \sim_i g_2$, then we have $f(g_1) = f(g_2)$, i.e. the strategy is a mapping on local states $f : L_{i\mathrm{E}} \to Act_i$.

The set of all uniform shared memoryless strategies of the set of agents A satisfying the above properties is denoted $UStr_A$. The set of all possible shared strategies is defined as $UStr \triangleq \bigcup_{A\subseteq Agt}^{A\neq\emptyset} UStr_A$.

We augment the definition of $\mathsf{sharing}$ (Definition 2.22) with an extra case for the epistemic modalities: $\mathsf{sharing}(\mathsf{K}_i\, \varphi) \triangleq \mathsf{sharing}(\mathsf{E}_A\, \varphi) \triangleq \mathsf{sharing}(\mathsf{D}_A\, \varphi) \triangleq \mathsf{sharing}(\mathsf{C}_A\, \varphi) \triangleq \mathsf{sharing}(\varphi)$ for all $i \in Agt$ and $A \subseteq Agt$. The definitions of *assignment* and *play* in SLK are analogous to the ones for SL (Definitions 2.23 and 2.26). The only difference is that they refer to uniform shared *memoryless* strategies and thus do not require translation (Definitions 2.24 and 2.25). We will omit them for conciseness.

4.1.3 Semantics

The semantics of SLK is defined with respect to interpreted systems (see Definition 2.5). For an interpreted system $\mathcal{I}$, a global state $g \in G$, and an assignment χ with $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(\chi)$, we write $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \varphi$ to indicate that the formula $\varphi \in SLK$ holds at g in $\mathcal{I}$ under χ. A formal definition follows.

Definition 4.3 (SLK Semantics)**.** Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i\in Agt}, I, h\right\rangle$ be an interpreted system and AP a set of propositional formulas. For all SLK formulas $\varphi \in SLK$, global states $g \in G$, and assignments $\chi \in Asg$ with $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(\chi)$, the modelling relation $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \varphi$ is inductively defined as follows:

1. $\mathcal{I}, \chi, g \models_{\text{SLK}} \top$.
2. $\mathcal{I}, \chi, g \models_{\text{SLK}} p$ iff $g \in h(p)$, with $p \in AP$.
3. For all formulas $\varphi, \varphi_1, \varphi_2 \in SLK$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\text{SLK}} \neg\varphi$ iff $\mathcal{I}, \chi, g \not\models_{\text{SLK}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1 \wedge \varphi_2$ iff $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1$ and $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_2$.
   (c) $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1 \vee \varphi_2$ iff $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1$ or $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_2$.
4. For a variable $x \in Var$ and a formula $\varphi \in SLK$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\text{SLK}} \langle\langle x \rangle\rangle\varphi$ iff there exists a strategy $f \in UStr_{\mathsf{sharing}(\varphi,x)}$ such that $\mathcal{I}, \chi[x \mapsto f], g \models_{\text{SLK}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\text{SLK}} [[x]]\varphi$ iff for all strategies $f \in UStr_{\mathsf{sharing}(\varphi,x)}$ it holds that $\mathcal{I}, \chi[x \mapsto f], g \models_{\text{SLK}} \varphi$.
5. For an agent $i \in Agt$, a variable $x \in Var$, and a formula $\varphi \in SLK$, it holds that $\mathcal{I}, \chi, g \models_{\text{SLK}} (i, x)\varphi$ iff $\mathcal{I}, \chi[i \mapsto \chi(x)], g \models_{\text{SLK}} \varphi$.
6. Let $\pi = \mathsf{play}(g, \chi)$. For all formulas $\varphi, \varphi_1, \varphi_2 \in SLK$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{X}\,\varphi$ iff $\mathcal{I}, \chi, \pi(1) \models_{\text{SLK}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{F}\,\varphi$ iff there exists $i \geq 0$ such that $\mathcal{I}, \chi, \pi(i) \models_{\text{SLK}} \varphi$;
   (c) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{G}\,\varphi$ iff for all $i \geq 0$ we have $\mathcal{I}, \chi, \pi(i) \models_{\text{SLK}} \varphi$;
   (d) $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1 \,\mathsf{U}\, \varphi_2$ iff there is an index $i \in \mathbb{N}$ such that $\mathcal{I}, \chi, \pi(i) \models_{\text{SLK}} \varphi_2$ and, for all indices $j \in \mathbb{N}$ with $0 \leq j < i$, it holds that $\mathcal{I}, \chi, \pi(j) \models_{\text{SLK}} \varphi_1$.
7. For an agent $i \in Agt$, a set of agents $A \subseteq Agt$, and a formula $\varphi \in SLK$, it holds that:
   (a) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{K}_i\,\varphi$ iff for all $g' \in G$ such that $g \sim_i g'$, it holds that $\mathcal{I}, \emptyset, g' \models_{\text{SLK}} \varphi$;
   (b) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{E}_A\,\varphi$ iff for all $g' \in G$ such that $g \sim_A^{\mathsf{E}} g'$, it holds that $\mathcal{I}, \emptyset, g' \models_{\text{SLK}} \varphi$;
   (c) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{D}_A\,\varphi$ iff for all $g' \in G$ such that $g \sim_A^{\mathsf{D}} g'$, it holds that $\mathcal{I}, \emptyset, g' \models_{\text{SLK}} \varphi$;
   (d) $\mathcal{I}, \chi, g \models_{\text{SLK}} \mathsf{C}_A\,\varphi$ iff for all $g' \in G$ such that $g \sim_A^{\mathsf{C}} g'$, it holds that $\mathcal{I}, \emptyset, g' \models_{\text{SLK}} \varphi$.

Although it is very similar, the SLK semantics differs from SL semantics (see Definition 2.29) in two aspects: *(i)* it is defined on uniform shared memoryless strategies and *(ii)* it supports epistemic operators. Both modifications are consequences of using incomplete information. Notice that we assume that agents are not aware of the strategy assignment. In fact, they are not even aware of their own strategy for the purposes of epistemic accessibility. This assumption is expressed by the empty assignments in the epistemic cases.

In order to complete the description of the SLK semantics, we define satisfiability and a model of an SLK sentence. Intuitively, a formula is satisfiable if it is true in some interpreted system.

Definition 4.4 (SLK Satisfiability)**.** We say that an interpreted system $\mathcal{I}$ is a *model* for an SLK sentence φ, in symbols $\mathcal{I} \models_{\text{SLK}} \varphi$, iff, for all initial global states $g_i \in I$, it holds that $\mathcal{I}, \emptyset, g_i \models_{\text{SLK}} \varphi$. More generally, we say that an interpreted system $\mathcal{I}$ is a *model* for an SLK sentence φ on a global state $g \in G$, in symbols $\mathcal{I}, g \models_{\text{SLK}} \varphi$, iff $\mathcal{I}, \emptyset, g \models_{\text{SLK}} \varphi$. An SLK sentence φ is *satisfiable* iff there is a model for it.

Finally, we define implication and equivalence between SLK formulas.

Definition 4.5 (SLK Implication and Equivalence)**.** Given two SLK formulas φ_1 and φ_2 with $\mathsf{free}(\varphi_1) = \mathsf{free}(\varphi_2)$, we say that φ_1 *implies* φ_2, in symbols $\varphi_1 \Rightarrow \varphi_2$, iff, for all interpreted systems $\mathcal{I}$, global states $g \in G$ and assignments $\chi \in Asg$ with $\mathsf{free}(\varphi) \subseteq \text{dom}(\chi)$, it holds that if $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_1$ then $\mathcal{I}, \chi, g \models_{\text{SLK}} \varphi_2$. We say that φ_1 is *equivalent* to φ_2, in symbols $\varphi_1 \equiv \varphi_2$, iff both $\varphi_1 \Rightarrow \varphi_2$ and $\varphi_2 \Rightarrow \varphi_1$ hold.

We will now consider two sample SLK formulas and explain why they do or do not hold in the toy model in Figure 3.1:

- $\mathcal{I}, \emptyset, g_{\mathrm{g}} \models_{\mathrm{SLK}} [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{X}\, p_2$. This formula expresses that *"whatever strategies the environment and player 1 use, there is a strategy for player 2, such that he will win in the next round"*. Intuitively, this is true. To see why it is formally true, let d be an arbitrary strategy for the environment and f an arbitrary strategy for player 1. Define a strategy h for player 2 as follows:

$$h(g_{\mathrm{g}}) \triangleq \begin{cases} \mathrm{r} & \text{if } f(g_{\mathrm{g}}) = \mathrm{s} \\ \mathrm{p} & \text{if } f(g_{\mathrm{g}}) = \mathrm{r} \\ \mathrm{s} & \text{if } f(g_{\mathrm{g}}) = \mathrm{p} \end{cases}$$

  We now need to show that $\mathcal{I}, \chi, g_{\mathrm{g}} \models_{\mathrm{SLK}} \mathsf{X}\, p_2$ where $\chi = \{(e, d), (x, f), (y, h), (\mathrm{E}, d), (1, f), (2, h)\}$. The resulting play is $\pi = g_{\mathrm{g}} g_2^{\omega}$ (player 2 always beats player 1 in the first round by the construction of h). The formula $\mathsf{X}\, p_2$ is true at g_{g} iff $\mathcal{I}, \chi, \pi(1) \models_{\mathrm{SLK}} p_2$. Since we have $\pi(1) = g_2 \in h(p_2)$ and d and f were arbitrary, we are done.

- $\mathcal{I}, \emptyset, g_{\mathrm{g}} \not\models_{\mathrm{SLK}} \langle\!\langle y\rangle\!\rangle[\![e]\!][\![x]\!](\mathrm{E}, e)(1, x)(2, y)\mathsf{X}\, p_2$. This formula expresses that *"there is a strategy for player 2, such that whatever strategies the environment and player 1 use, he will win in the next round"*. Again, our intuition tells us that this formula should be false. To see why this is indeed false, let h an arbitrary strategy for player 2. We pick some strategy d for the environment and define a strategy f for player 1 as follows:

$$f(g_{\mathrm{g}}) \triangleq \begin{cases} \mathrm{r} & \text{if } h(g_{\mathrm{g}}) = \mathrm{s} \\ \mathrm{p} & \text{if } h(g_{\mathrm{g}}) = \mathrm{r} \\ \mathrm{s} & \text{if } h(g_{\mathrm{g}}) = \mathrm{p} \end{cases}$$

  We now need to show that $\mathcal{I}, \chi, g_{\mathrm{g}} \not\models_{\mathrm{SLK}} \mathsf{X}\, p_2$ where $\chi = \{(e, d), (x, f), (y, h), (\mathrm{E}, d), (1, f), (2, h)\}$. The resulting play is $\pi = g_{\mathrm{g}} g_1^{\omega}$ (player 1 always beats player 2 in the first round by the construction of f). The formula $\mathsf{X}\, p_2$ is true at g_{g} iff $\mathcal{I}, \chi, \pi(1) \models_{\mathrm{SLK}} p_2$. Since we have $\pi(1) = g_1 \notin h(p_2)$ and h was arbitrary, we are done.

4.1.4 Comparison with Strategy Logic

Let us now reiterate the main differences between the two logics. SLK differs from SL in the following ways (please refer to Subsection 2.2.5 for more information about the original SL):

1. **Imperfect recall.** Intuitively, SLK agents have no memory of the past and make their decisions based on the current states only. Formally, SLK strategies are mappings from *local* states to actions ($f : L_{i\mathrm{E}} \to \mathit{Act}_i$). Compare this with ATL and SL strategies (see Definitions 2.16 and 2.21), which map non-empty finite sequences of *global* states (tracks) to actions ($f : \mathit{Trk} \to \mathit{Act}_i$).

2. **Incomplete information.** Agents do not have complete knowledge of the whole system in SLK, i.e. they do not "see" other agents' variables. This puts the uniformity constraint on strategies (see Definition 4.2) so that they could be executed by agents (see Subsection 4.1.2). As explained in Chapter 3, SL is undecidable under perfect recall semantics with incomplete information.

3. **Epistemic modalities.** SLK supports epistemic modalities expressing agents' knowledge. The four epistemic modalities supported by SLK are individual knowledge $\mathsf{K}_i\, \varphi$, group knowledge $\mathsf{E}_A\, \varphi$, distributed knowledge $\mathsf{D}_A\, \varphi$, and common knowledge $\mathsf{C}_A\, \varphi$. See Subsection 2.2.6 for more details about epistemic modalities.

4. **Underlying framework.** SLK is based on interpreted systems whereas SL was *originally* defined on concurrent game structures. Note that we redefined SL on interpreted systems in Subsection 2.2.5 so that it would be easier to compare the two logics. As explained in Subsection 2.1.3, interpreted systems are more natural for expressing incomplete information. Both frameworks are defined in Section 2.1.

While epistemic modalities dramatically increase the expressiveness of SLK, imperfect recall puts heavy constraints on agents' behaviour. The impact of no memory is discussed in the next subsection.

4.1.5 Limitations

As we explained in Section 3.1, SLK solves the undecidability of SL under incomplete information using imperfect recall semantics. Effectively, we are removing agents' memory and forcing them to base their decisions purely on their current local states. While memoryless strategies are easier to reason about and more compact, they are also less powerful and non-behavioural. We will now explain these limitations. For a start, we show that some simple properties may not be achievable using memoryless strategies. Consider the toy model in Figure 3.1 and the following SLK specification:

$$\langle\langle e\rangle\rangle\langle\langle x\rangle\rangle\langle\langle y\rangle\rangle(\mathrm{E}, e)(1, x)(2, y)[\mathsf{X}\ (\neg p_1 \wedge \neg p_2) \wedge \mathsf{X}\,\mathsf{X}\ (p_1 \vee p_2)]$$

The rough meaning of the formula is *"There exist strategies for the two players such that they draw first and then one of them wins"*. Clearly, there exist memoryful strategies which satisfy the specification above (starting from the initial state g_{g}): The agents simply perform the same action in the first round (e.g. rock-rock) and different actions in the second round (e.g. rock-scissors). Formally, the memoryful strategies f_1, f_2 are:

$$f_1(g_{\mathrm{g}}) \triangleq \mathrm{r} \qquad f_2(g_{\mathrm{g}}) \triangleq \mathrm{r}$$
$$f_1(g_{\mathrm{g}}g_{\mathrm{g}}) \triangleq \mathrm{r} \qquad f_2(g_{\mathrm{g}}g_{\mathrm{g}}) \triangleq \mathrm{s}$$

However, there are *no memoryless strategies* that would fulfil the specification. The reason is that since both agents perform an action $a \in \{\mathrm{r}, \mathrm{p}, \mathrm{s}\}$ in the first round, they have to do it again in the next round because their local state has not changed. Hence, they will keep drawing forever.

We can see that memoryless strategies have less power than memoryful strategies. This can be useful in certain scenarios. Consider the situation where we are synthesising the behaviour of a simple hardware device (e.g. a thermostat) with a fixed amount of memory. In this case, we consider each possible memory settings of the device to be one local state and allow arbitrary transitions between them. Informally, we let the agent decide what it wants to remember. It is appropriate to treat the agent as memoryless since its memory is already encoded in its local state space. We then try to synthesise a memoryless strategy for it. If we succeed, then the agent's fixed-size memory is sufficient to achieve the goal. Hence, SLK can be used to answer questions like *"Does agent a have enough memory to enforce φ?"*.

On the other hand, there are many scenarios when it is clearly inappropriate to assume that the agent has no memory. Consider the following security protocol specification:

$$\langle\langle e\rangle\rangle[\![x]\!](\mathrm{E}, e)(\mathrm{intruder}, x)\mathsf{G}\,\neg\mathsf{K}_{\mathrm{intruder}}\,\mathit{password}$$

It asserts that there is some strategy e for the environment such that whatever strategy x the intruder uses, they will never know the password. The major flaw with this specification is that it assumes that the intruder has no memory, i.e. it *underestimates* their power. Hence, we can be certain about a "yes" answer only if we know exactly what they are capable of, which is rarely the case. Conversely, if the answer is "no", it does not necessarily mean that there is no memoryful strategy for the environment that would ensure security. This simple example demonstrates that SLK is not appropriate for the verification of security protocols[1].

As we pointed out in Section 3.1, imperfect recall does not address the problem of non-behavioural strategies, in which an agent's action depends on another agent's action in another counterfactual play. Consider again the toy model in Figure 3.1 and the following SLK specification:

$$\langle\langle e\rangle\rangle(\mathrm{E}, e)[\![y]\!][\![z]\!]\langle\langle x\rangle\rangle[((1, x)(2, y)\mathsf{X}\,p_1) \leftrightarrow ((1, y)(2, z)\mathsf{X}\,p_2)]$$

This formula is more complicated than the previous ones. Its meaning roughly is as follows: *"For all strategies y, z there is a strategy x, such that x beats y iff z beats y"*. Given this, it should be clear that this formula is true because we can always set $x = z$. Now the question is, how do we determine the strategy x? The action of player 1 in $(1, x)(2, y)\mathsf{X}\,p_1$ depends on the action of player 2 in $(1, y)(2, z)\mathsf{X}\,p_2$. In other words, the strategy of player 1 depends on the strategy of player 2 in another counterfactual play, i.e. it is *non-behavioural.* While the formula above might seem quite convoluted, it is precisely this ability to express complicated interdependencies of agents' behaviour why SL was introduced in the first place. Please refer to [75] if you want to know more about behavioural and non-behavioural fragments of SL.

[1]Nevertheless, we will do exactly that in Subsection 6.4.1.

4.2 Model Checking

In this section, we focus on model checking SLK. Recall that the model checking problem for SLK is as follows: Given a system S represented by an interpreted system $\mathcal{I}_S$ and a property P expressed as an SLK formula $\varphi_P \in SLK$, we want to determine whether the formula is true in the model:

$$\mathcal{I}_S \overset{?}{\models}_{\text{SLK}} \varphi_P$$

We provide a model checking algorithm for SLK which calculates the set $\|\varphi_P\|_{\mathcal{I}_S}$ of global states of $\mathcal{I}_S$ in which the formula φ_P holds. The problem above is then decided by checking if all initial states I of $\mathcal{I}_S$ satisfy φ_P:

$$I \overset{?}{\subseteq} \|\varphi_P\|_{\mathcal{I}_S}$$

We first discuss the complexity of the problem. Then we present an algorithm which admits an efficient symbolic implementation.

4.2.1 Complexity

Before developing an algorithm for model checking SLK, we consider the complexity of the SLK model checking (decision) problem:

> Given an interpreted system $\mathcal{I}$, a global state $g \in G$, and an SLK formula $\varphi \in SLK$, determine whether $\mathcal{I}, \emptyset, g \models_{\text{SLK}} \varphi$.

We claim that the problem can be solved in a polynomial amount of space (PSPACE) with respect to both the size of the interpreted system[2] $|\mathcal{I}|$ and the size of the formula $|\varphi|$. The idea is to determine the problem recursively with a function that accepts a global state $g' \in G$ and an assignment $\chi' \in Asg$ as input. Since both g' and χ' require only a polynomial amount of space and there will be at most $|\varphi|$ nested calls to the function, we obtain the desired complexity.

Theorem 4.1. The SLK model checking problem is PSPACE with respect to both the size of the model $|\mathcal{I}|$ and the size of the formula $|\varphi|$.

Proof. An arbitrary state $g' \in G$ can be encoded using $\lceil \log_2 |G| \rceil$ Boolean variables (how this can be done is explained in Subsection 2.3.2). An arbitrary uniform shared memoryless strategy $f_A : G \to Act_A$ for a set of agents $A \subseteq Agt$ can be stored using $\left\lceil \log_2 \left| \left[\bigcap_{i \in A} Act_i \right]^G \right| \right\rceil \leq |G| \times \lceil \log_2 |\max_{i \in A} Act_i| \rceil$ space. Note that we assumed in Definition 2.22 that every variable is quantified at most once within a formula. Therefore, we do not need to remember the full assignment on $Var \cup Agt$ as the entries for agents are merely copies of the entries for the variables they are bound to. Hence, an arbitrary assignment $\chi' \in Asg$ can be stored using at most

$$\underbrace{|\mathsf{vars}(\varphi)| \times (|G| \times \lceil \log_2 |\max_{i \in A} Act_i| \rceil + 1)}_{\text{partial mapping from variables in } \varphi \text{ to strategies}} + \underbrace{|Agt| \times \lceil \log_2 |\mathsf{vars}(\varphi)| \rceil}_{\substack{\text{“pointers” for agents into} \\ \text{the variable assignment}}} = O(|\mathcal{I}|^2 \times |\varphi|)$$

space. The semantics of SLK (Definition 4.3) can be easily transformed into a function CHECK(φ', g', χ'). The important cases are (we omit the other cases for conciseness):

- CHECK$(\langle\langle x \rangle\rangle \varphi'', g', \chi')$. We store the current strategy for the variable x, $s := \chi'(x)$. Then we perform a `for` loop over all possible strategies $\chi'(x) := f_{\mathsf{sharing}(\varphi, x)}$ and call CHECK(φ'', g', χ'). After the loop we restore the current strategy for the variable x, $\chi'(x) := s$. If any of the calls within the loop succeeded, we return “yes”. Otherwise, we return “no”.

  The temporary storage s for one strategy and some counter variables for the loop fit within the $O(|\mathcal{I}|^2 \times |\varphi|)$ space bound.

- CHECK$(\mathsf{X}\, \varphi'', g', \chi')$. Given g' and χ'', it is easy to calculate the successor g'' (by calculating $\pi(1)$ of $\pi = \mathsf{play}(\chi', g')$ as in Definition 2.26). We then call CHECK(φ'', g'', χ').

  $O(|\mathcal{I}|^2 \times |\varphi|)$ space will be sufficient for calculating the successor.

[2]To be previse, we define $|\mathcal{I}| = |G| + |Agt| + |Act|$.

- CHECK($\varphi'' \mathsf{U}\, \varphi''', g', \chi'$). We introduce a counter i and perform the following (pseudocode):

  ```
  for $i := 0$ to $|G|$ do
      if CHECK($\varphi''', g', \chi'$) then
          return "yes"
      else if $\neg$CHECK($\varphi'', g', \chi'$) then
          return "no"
      end if
      $g' :=$ SUCCESSOR($g', \chi'$)
  end for
  return "no"
  ```

  We traverse the path starting from g' as long as φ'' holds. If we reach a state where φ''' holds, the formula $\varphi'' \mathsf{U}\, \varphi'''$ holds so we return "yes". If we do not reach such a state within $|G|$ steps, we must be in a cycle (since there are $|G|$ global states) where φ''' never holds. Therefore, we return "no".

  The extra counter (and perhaps some auxiliary variables) will surely fit within $O(|\mathcal{I}|^2 \times |\varphi|)$ space.

The depth of the call stack will be at most $|\varphi|$. Hence, if we reuse space, CHECK($\varphi, g, \emptyset$) will use at most $O(|\mathcal{I}|^2 \times |\varphi|^2)$ space. Our claim follows. □

The model checking complexity of ATL* with imperfect recall is PSPACE-COMPLETE [23]. Therefore, it is tempting to immediately conclude that the problem for SLK is also PSPACE-COMPLETE. However, there is an important difference between ATL* and SLK semantics: ATL* assigns memoryless strategies only to the existentially quantified agents (see Definition 2.18), whereas SLK assigns memoryless strategies to all agents. Informally, an ATL* expression $\langle\langle A\rangle\rangle\psi$ means that *"There exist memoryless strategies for agents A such that no matter what the other agents do, ψ will be true"*. The SLK formula $[\langle\langle x_i\rangle\rangle(i, x_i)]_{i\in A}\, [[\![y_i]\!](i, y_i)]_{i\in Agt\setminus A}\, \varphi$ has a slightly different meaning: *"There exist memoryless strategies for agents A, such that for all memoryless strategies of the other agents, ψ will be true"*. Intuitively, SLK restricts the universally quantified agents more than ATL* does[3].

Nevertheless, we conjecture that the SLK model checking problem is indeed PSPACE-COMPLETE. We believe that this could be shown by reducing the problem of evaluating a Quantified Boolean Formula[4], which is PSPACE-HARD [45], to the SLK model checking problem.

4.2.2 Algorithm

The model checking algorithm $\mathsf{SAT}_{\mathrm{SLK}}$ for SLK, which calculates the set of global states in which a given formula is true, is a modification of the existing model checking algorithm for ATLK[5] used by MCMAS [65, 81]. It differs in two ways:

1. It has an extra parameter, which represents the *binding* of agents to variables. When model checking a formula, we start with an empty binding and augment it whenever an agent binding operator $(i, x)\varphi$ is encountered.

2. Unlike the original algorithm, which merely returns the set of states in which a given formula holds, the modified algorithm returns a set of *extended states*. Intuitively, an extended state is a pair of *(i)* a global state and *(ii)* a variable assignment (mapping variables to strategies) subject to which the formula holds in that state.

We will now define the aforementioned concepts of a binding and variable assignment more formally. For simplicity, we will fix *Var* to always be the set of variables quantified in the SLK formula we are considering (e.g. if the formula to be checked is $\varphi = \langle\langle x\rangle\rangle[\![y]\!](a, x)(b, y)\mathsf{X}\, p$, then we set $Var = \mathsf{vars}(\varphi) = \{x, y\}$). This will allow us to define variable assignments as total functions, which will make the theory and proofs much simpler.

[3]Note that this subtle difference in semantics has no effect on the relationship between SL[1G] and ATL* with perfect recall (SL[1G] strictly subsumes ATL* [72]). Imposing memoryful strategies on agents does not constrain their behaviour in any way.

[4]A *Quantified Boolean Formula* has the form $\mathsf{Q}_0 x_0 \cdots \mathsf{Q}_{n-1} x_{n-1}.\, E(x_0, \ldots, x_{n-1})$ where E is a propositional formula over propositional variables $x_0, \ldots, x_{n-1}$ and for all $0 \le i < n$, Q_i is an existential quantifier $\exists$ or a universal quantifier $\forall$.

[5]A similar model checking algorithm for CTL is shown in Definition 2.12.

Definition 4.6 (Bindings)**.** Let $\mathcal{I}$ be an interpreted system. Then a *binding* is a partial function $b : Agt \rightharpoonup Var$ which maps agents in its domain to variables. $Bnd \triangleq Agt \rightharpoonup Var$ denotes the set of all bindings.

Definition 4.7 (Variable Assignments)**.** Let $\mathcal{I}$ be an interpreted system. Then a *variable assignment* is a function $v : Var \to UStr$ which maps variables in its domain to uniform shared memoryless strategies. $VAsg \triangleq Var \to UStr$ denotes the set of all variable assignments.

Note that variable assignments (Definition 4.7) are also assignments (Definition 2.23 modified for SLK strategies as explained in Subsection 4.1.2), i.e. $VAsg \subseteq Asg$.

As we have already explained, an extended state is a pair of a *(i)* global state and *(ii)* a variable assignment subject to which a formula is true in that state. For example, $(g_1, \{(x, f), (y, g)\})$ is an extended state which means roughly: *"(the formula is true in) global state g_1 when agents bound to variables x and y act according to the strategies f and g respectively."*. A formal definition follows.

Definition 4.8 (Extended States)**.** Let $\mathcal{I}$ be an interpreted system, $g \in G$ a global state and $v \in VAsg$ a variable assignment. Then an *extended state* is a pair $\langle g, v\rangle \in G \times VAsg$. $Ext \triangleq G \times VAsg$ denotes the set of all extended states.

The meaning of the variable assignment in an extended state is probably slightly unclear right now. Intuitively, an extended state $\langle g, v\rangle \in Ext$ *guarantees* a formula $\varphi \in SLK$ iff all assignments which agree with v make the formula true in the state g. Since this explanation is still quite vague, we define the notion of *guarantee* more formally.

Definition 4.9 (Guarantee)**.** Let $\mathcal{I}$ be an interpreted system, $\langle g, v\rangle \in Ext$ an extended state, $b \in Bnd$ a binding, and $\varphi \in SLK$ an SLK formula with $\mathsf{free}(\varphi) \cap Agt \subseteq \mathrm{dom}(b)$. We say that $\langle g, v\rangle$ *guarantees* φ in $\mathcal{I}$ under b iff for the assignment $\chi_v \in Asg$ defined as:

$$\chi_v \triangleq v \cup \{(a, v(b(a))) \mid a \in \mathrm{dom}(b)\}$$

we have $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \varphi$. We write this as $\mathcal{I}, \langle g, v\rangle, b \vdash \varphi$.

Now that we have covered the basic structures, we can define the concepts of negation and predecessors of extended states. These will be necessary for the model checking algorithm presented at the end of this subsection. Let us start with negation. Assume that we have calculated the set of extended states E which guarantee the formula φ under a binding b. We want to find the set E' of extended states which guarantee the formula $\neg\varphi$. Intuitively, E' should contain all extended states that somehow disagree with E. E' is calculated as follows:

$$E' = Ext \setminus E$$

We shall now prove that our claim is correct. We assume that E is the set of all extended states that guarantee φ under b and show that $Ext \setminus E$ is the set of all extended states that guarantee $\neg\varphi$ under b.

Lemma 4.1. Let $\mathcal{I}$ be an interpreted system, $b \in Bnd$ a binding, and $\varphi \in SLK$ an SLK formula with $\mathsf{free}(\varphi) \cap Agt \subseteq \mathrm{dom}(b)$. Let $E \subseteq Ext$ be the set of all extended states which guarantee φ in $\mathcal{I}$ under b. Then $Ext \setminus E$ is the set of all extended states which guarantee $\neg\varphi$ in $\mathcal{I}$ under b.

Proof. Let $E' \subseteq Ext$ be the set of all extended states which guarantee $\neg\varphi$ in $\mathcal{I}$ under b. We show that $E' = Ext \setminus E$.

$\Rightarrow$: Take an arbitrary extended state $\langle g, v\rangle \in E'$. Since $\langle g, v\rangle$ guarantees $\neg\varphi$, we have $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \neg\varphi$. By SLK semantics (Definition 4.3), $\mathcal{I}, \chi_v, g \not\models_{\text{SLK}} \varphi$. Thus $\langle g, v\rangle \notin E$, so we have $\langle g, v\rangle \in Ext \setminus E$.

$\Leftarrow$: Take an arbitrary extended state $\langle g, v\rangle \in Ext \setminus E$. Since $\langle g, v\rangle$ does not guarantee φ (otherwise, we would have $\langle g, v\rangle \in E$), we have $\mathcal{I}, \chi_v, g \not\models_{\text{SLK}} \varphi$. By SLK semantics (Definition 4.3), $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \neg\varphi$. Thus, $\langle g, v\rangle \in Ext$ guarantees $\neg\varphi$.

□

Having covered negation, it remains to explain how to calculate the set of previous extended states, i.e. given a set of extended states $E \subseteq \mathit{Ext}$ which guarantee φ under a binding b, determine the set of extended states $E' \subseteq \mathit{Ext}$ which guarantee $\mathsf{X}\,\varphi$ under the same binding.

We first define the transition relation on global states implied by a binding and a variable assignment. Intuitively, there is a transition from state $g_1 \in G$ to a state $g_2 \in G$ (given the binding and variable assignment) if there exists a joint action between them such that each agent acts according to the strategy assigned to the variable it is bound to.

Definition 4.10 (Implied Transition Relation)**.** Let $\mathcal{I}$ be an interpreted system, $g_1, g_2 \in G$ two global states, $b \in \mathit{Bnd}$ a binding, and $v \in \mathit{VAsg}$ a variable assignment. Then the *transition relation* $\to_v^b \subseteq G \times G$ *implied* by b and v is defined by $g_1 \to_v^b g_2$, iff $\mathrm{dom}(b) = \mathit{Agt}$ and there exists a joint action $a \in \mathit{Act}$ such that $t(g_1, a) = g_2$, and, for all agents $i \in \mathit{Agt}$, it holds that $a_i(a) = v(b(i))(g_1)$.

We are now ready to explain how the set of previous extended states is calculated. Intuitively, given an extended state $\langle g, v\rangle \in \mathit{Ext}$, the previous extended states are pairs $\langle g', v\rangle$ where g' is the successor of g when all agents act according to their strategies in v.

Definition 4.11 (Previous Extended States)**.** Let $\mathcal{I}$ be an interpreted system, $E \subseteq \mathit{Ext}$ a set of extended states, and $b \in \mathit{Bnd}$ a binding such that $\mathrm{dom}(b) = \mathit{Agt}$. Then the function $\mathsf{pre} : 2^{\mathit{Ext}} \times \mathit{Bnd} \to 2^{\mathit{Ext}}$, which returns the set of *previous extended states*, is defined as $\mathsf{pre}(E, b) \triangleq \big\{\langle g, v\rangle \in \mathit{Ext} \bigm| \exists g' \in G.\, \langle g', v\rangle \in E \wedge g \to_v^b g'\big\}$.

Again, we will show that the function pre is correct. We assume that E is the set of all extended states which guarantee φ under b and show that $\mathsf{pre}(E)$ is the set of all extended states which guarantee $\mathsf{X}\,\varphi$ under b.

Lemma 4.2. Let $\mathcal{I}$ be an interpreted system, $b \in \mathit{Bnd}$ a binding with $\mathrm{dom}(b) = \mathit{Agt}$, and $\varphi \in \mathit{SLK}$ an SLK formula. Let $E \subseteq \mathit{Ext}$ be the set of all extended states which guarantee φ in $\mathcal{I}$ under b. Then $\mathsf{pre}(E) \subseteq \mathit{Ext}$ is the set of all extended states which guarantee $\mathsf{X}\,\varphi$ in $\mathcal{I}$ under b.

Proof. Let $E' \subseteq \mathit{Ext}$ be the set of all extended states which guarantee $\mathsf{X}\,\varphi$ in $\mathcal{I}$ under b. We show that $\mathsf{pre}(E) = E'$.

$\Rightarrow$: Take an arbitrary extended state $\langle g, v\rangle \in \mathsf{pre}(E, b)$. By construction, there is a global state $g' \in G$ such that $\langle g', v\rangle \in E$ and $g \to_v^b g'$. Since $\langle g', v\rangle \in E$ guarantees φ, we have $\mathcal{I}, \chi_v, g' \models_{\text{SLK}} \varphi$. The implied transition relation implies $g' = t(g, a)$ where $a_i(a) = v(b(i))(g)$ for all agents $i \in \mathit{Agt}$. This can be also rewritten as $g' = t(g, \langle \chi_v(a)(g) : a \in \mathit{Agt}\rangle)$ because $\chi_v(i) = v(b(i))$ for all agents $i \in \mathrm{dom}(b) = \mathit{Agt}$.

We want to show that $\langle g, v\rangle$ guarantees $\mathsf{X}\,\varphi$, i.e. $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \mathsf{X}\,\varphi$. This is the case iff $\mathcal{I}, \chi_v, \pi(1) \models_{\text{SLK}} \varphi$ where $\pi = \mathsf{play}(\chi_v, g)$ (Definition 4.3). From Definition 2.26, we get $\pi(1) = t(g, \langle \chi(a)(g) : a \in \mathit{Agt}\rangle)$, so $\pi(1) = g'$. Since we have already shown $\mathcal{I}, \chi_v, g' \models_{\text{SLK}} \varphi$, we have $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \mathsf{X}\,\varphi$ as required.

$\Leftarrow$: Take an arbitrary extended state $\langle g, v\rangle \in E'$. Thus, we have $\mathcal{I}, \chi_v, g \models_{\text{SLK}} \mathsf{X}\,\varphi$. By Definition 4.3, this means $\mathcal{I}, \chi_v, \pi(1) \models_{\text{SLK}} \varphi$ where $\pi = \mathsf{play}(\chi_v, g)$. From Definition 2.26, we get $\pi(1) = t(g, \langle \chi_v(a)(g) : a \in \mathit{Agt}\rangle)$. This can be equivalently written as $\pi(1) = t(g, a)$ where $a_i(a) = \chi_v(a)(g) = v(b(i))(g)$ for all agents $i \in \mathit{Agt}$.

As $\mathrm{dom}(b) = \mathit{Agt}$ by assumption, we have $g \to_v^b \pi(1)$. Moreover, since $\mathcal{I}, \chi_v, \pi(1) \models_{\text{SLK}} \varphi$, we have $\langle \pi(1), v\rangle \in E$ (because it guarantees φ). Therefore, we have $\langle g, v\rangle \in \mathsf{pre}(E, b)$ as required.

□

Finally, we have all the ingredients to define the model checking algorithm $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\cdot,\cdot)$ for SLK.

Definition 4.12 (SLK Model Checking Algorithm)**.** Let $\mathcal{I}$ be an interpreted system, $\varphi \in \mathit{SLK}$ an SLK formula and $b \in \mathit{Bnd}$ a binding, such that $\mathsf{free}(\varphi) \cap \mathit{Agt} \subseteq \mathrm{dom}(b)$. Then the model checking function $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}} : \text{SLK} \times \mathit{Bnd} \to 2^{\mathit{Ext}}$ is inductively defined as follows:

1. $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\top, b) \triangleq \mathit{Ext}$.

2. $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(p, b) \triangleq \{\langle g, v\rangle \mid g \in h(p)\}$, with $p \in \mathit{AP}$.

3. For all formulas $\varphi, \varphi_1, \varphi_2 \in \mathit{SLK}$, it is defined as:

   (a) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\varphi, b) \triangleq \mathit{Ext} \setminus \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b)$;

   (b) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1 \wedge \varphi_2, b) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1, b) \cap \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_2, b)$;

   (c) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1 \vee \varphi_2, b) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1, b) \cup \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_2, b)$.

4. For an agent $i \in \mathit{Agt}$, a variable $x \in \mathit{Var}$, and a formula $\varphi \in \mathit{SLK}$, $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}((i,x)\varphi, b) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b[i \mapsto x])$.

5. For a variable $x \in \mathit{Var}$ and an SLK formula $\varphi \in \mathit{SLK}$, it is defined as:

   (a) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\langle\!\langle x \rangle\!\rangle\varphi, b) \triangleq \big\{\langle g, v\rangle \;\big|\; \exists f \in \mathit{UStr}_{\mathsf{sharing}(\varphi,x)}.\, \langle g, v[x \mapsto f]\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b)\big\}$;

   (b) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}([\![x]\!]\varphi, b) \triangleq \big\{\langle g, v\rangle \;\big|\; \forall f \in \mathit{UStr}_{\mathsf{sharing}(\varphi,x)}.\, \langle g, v[x \mapsto f]\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b)\big\}$.

6. For all formulas $\varphi, \varphi_1, \varphi_2 \in \mathit{SLK}$, it is defined as:

   (a) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{X}\,\varphi, b) \triangleq \mathsf{pre}(\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b)\,, b)$;

   (b) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{F}\,\varphi, b) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\top\,\mathsf{U}\,\varphi, b)$;

   (c) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{G}\,\varphi, b) \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\mathsf{F}\,\neg\varphi, b)$;

   (d) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1\,\mathsf{U}\,\varphi_2, b) \triangleq \mathrm{lfp}_X\big[\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_2, b) \cup \big(\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi_1, b) \cap \mathsf{pre}(X, b)\big)\big]$.

7. For an agent $i \in \mathit{Agt}$, a set of agents $A \subseteq \mathit{Agt}$, and a formula $\varphi \in \mathit{SLK}$, it is defined as:

   (a) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{K}_i\,\varphi, b) \triangleq \mathit{Ext} \setminus \big\{\langle g, v\rangle \in \mathit{Ext} \;\big|\; \exists\,\langle g', v'\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\varphi, \emptyset)\,.g' \sim_i g\big\}$;

   (b) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{E}_A\,\varphi, b) \triangleq \mathit{Ext} \setminus \big\{\langle g, v\rangle \in \mathit{Ext} \;\big|\; \exists\,\langle g', v'\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\varphi, \emptyset)\,.g' \sim^{\mathsf{E}}_A g\big\}$;

   (c) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{D}_A\,\varphi, b) \triangleq \mathit{Ext} \setminus \big\{\langle g, v\rangle \in \mathit{Ext} \;\big|\; \exists\,\langle g', v'\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\varphi, \emptyset)\,.g' \sim^{\mathsf{D}}_A g\big\}$;

   (d) $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\mathsf{C}_A\,\varphi, b) \triangleq \mathit{Ext} \setminus \big\{\langle g, v\rangle \in \mathit{Ext} \;\big|\; \exists\,\langle g', v'\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\neg\varphi, \emptyset)\,.g' \sim^{\mathsf{C}}_A g\big\}$.

The correctness of the algorithm is asserted in the following theorem.

Theorem 4.2. Let $\mathcal{I}$ be an interpreted system and $\varphi \in \mathit{SLK}$ an SLK sentence. Then the set of all states at which φ holds is: $\{g \in G \mid \mathcal{I}, \emptyset, g \models_{\text{SLK}} \varphi\} = \big\{g \in G \;\big|\; \exists v \in \mathit{VAsg}.\, \langle g, v\rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, \emptyset)\big\}$.

Proof (Sketch). We prove by induction that for an arbitrary interpreted system $\mathcal{I}$, SLK formula $\varphi \in \mathit{SLK}$, and binding $b \in \mathit{Bnd}$ such that $\mathsf{free}(\varphi) \subseteq \mathrm{dom}(b)$, $\mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\varphi, b)$ is the set of all extended states that guarantee φ in $\mathcal{I}$ under b. The two important cases were proved in Lemmas 4.1 (negation) and 4.2 (temporal step).

Since the topmost formula φ is a sentence, it will either hold, or not hold in each state (regardless of the variable assignment). Therefore, we existentially quantify over variable assignments. □

Observe that the SLK model checking algorithm is decidable because the sets *UStr*, *Var*, *VAsg*, *Agt*, *Bnd*, G, and, consequently, *Ext* are finite. An efficient symbolic implementation of the algorithm using BDDs is presented in Subsection 4.2.4.

4.2.3 Strategy Synthesis

As we have explained in Subsection 4.1.5, one of the main limitations of SLK are *non-behavioural strategies*, which depend on counterfactual scenarios. Consequently, SLK strategies are very difficult[6] to synthesise. To see why this is the case, consider the SLK sentence $\varphi \triangleq [\![x]\!][\![y]\!]\langle\!\langle z \rangle\!\rangle\psi$. Assume that φ holds at a particular state $g \in G$ in an interpreted system $\mathcal{I}$, i.e. $\mathcal{I}, g \models_{\text{SLK}} \varphi$. We would now like to synthesise a uniform shared strategy $f_z : G \to \mathit{Act}_z$ for the variable z depending on the uniform shared strategies $f_x : G \to \mathit{Act}_x$ and $f_y : G \to \mathit{Act}_y$ for the variables x and y respectively where $\mathit{Act}_v \triangleq \mathit{Act}_{\mathsf{sharing}(\psi,v)}$

[6]Unlike perfect recall SL strategies, which cannot be synthesised in general due to being non-behavioural, imperfect recall SLK strategies can always be synthesised. This should not be surprising given that the set of uniform strategies is *finite* for a given interpreted system and SLK formula. Hence, we can explicitly enumerate all possible combinations of strategies.

for $v \in \mathit{Var}$. If SLK strategies were behavioural, there *would* exist a mapping m_1 (also referred to as *elementary dependence map* [75]) from the next actions of f_x and f_y in g to the next action of f_z in g:

$$m_1 : \underbrace{G \to (\mathit{Act}_x \times \mathit{Act}_y \to \mathit{Act}_z)}_{(g, f_x(g), f_y(g)) \mapsto f_z(g)}$$

There would be at most $|\mathit{Act}_x| \times |\mathit{Act}_y|$ possible inputs to m_1 to determine $f_z(g)$ as it depends only on $f_x(g)$ and $f_y(g)$. Unfortunately, such a mapping does not exist in general because SLK strategies are non-behavioural. Instead, a more general mapping m_2 (also referred to as *dependence map* [75]) from strategies f_x and f_y to the strategy f_z must be considered:

$$m_2 : \underbrace{(G \to \mathit{Act}_x) \times (G \to \mathit{Act}_y) \to (G \to \mathit{Act}_z)}_{(f_x, f_y, g) \mapsto f_z(g)}$$

Informally, determining $f_z(g)$ requires the same amount of information as constructing the whole strategy f_z. In order to synthesise the action $f_z(g)$ or the strategy f_z, we possibly need to know the complete strategies f_x and f_y. More importantly, the maximum number of *entries* in the mappings m_1 and m_2 are:

$$m_1 : |G| \times |\mathit{Act}_x| \times |\mathit{Act}_y| \qquad\qquad m_2 : |G| \times |\mathit{Act}_x|^{|G|} \times |\mathit{Act}_y|^{|G|}$$

Assume that the interpreted system $\mathcal{I}$ is quite small and has $|G| = 10$ global states and $|\mathit{Act}_x| = |\mathit{Act}_y| = |\mathit{Act}_z| = 10$ actions. While the mapping m_1 for *behavioural* strategies would require at most 1000 entries of the form $(g, f_x(g), f_y(g)) \mapsto f_z(g)$, the mapping m_2 for *non-behavioural* strategies might have up to 10^{21} entries of the form $(f_x, f_y, g) \mapsto f_z(g)$. Furthermore, if we encode each output of f_z using only $\lceil \log_2 |\mathit{Act}_z| \rceil = 3$ bits and store the whole mapping in a large array, m_1 will use at most 375 *bytes* while m_2 might need up to 375 *exabytes*. To put this number into perspective, the total amount of stored information in the whole world was 295 exabytes (optimally compressed) in 2007 [47]. This example demonstrates the high impracticality of general SLK strategy synthesis on any non-trivial models.

While SLK strategy synthesis is infeasible in general, it can be performed efficiently on certain types of formulas. We will now explain the concepts of *witness* and *counterexample strategies* and describe how these can be synthesised using the model checking algorithm discussed in Subsection 4.2.2. Let $\mathcal{I}$ be an interpreted system, $g \in G$ a global state, and $\varphi_w \triangleq \langle\langle x\rangle\rangle\psi_w$ and $\varphi_c \triangleq [[y]]\psi_c$ two SLK sentences. Furthermore, assume that φ_w holds at g while φ_c does not, i.e. $\mathcal{I}, \emptyset, g \models_{\text{SLK}} \varphi_w$ and $\mathcal{I}, \emptyset, g \not\models_{\text{SLK}} \varphi_c$. By SLK semantics (see Definition 4.3), there is a memoryless uniform shared strategy f_w for x which makes ψ_w true at g. Conversely, there must be a memoryless shared strategy f_c for y which makes ψ_c false at g. f_w and f_c are referred to as a *witness* and a *counterexample strategy* respectively. Intuitively, the strategy f_w is a "witness" to φ_w being true at g while f_c is a "counterexample" for φ_c at g. A slightly more general form of the two concepts is provided in the following definition.

Definition 4.13 (Witness and Counterexample Strategies)**.** Let $\mathcal{I}$ be an interpreted system, $g \in G$ a global state, and $\varphi_w \triangleq \langle\langle x_0\rangle\rangle \dots \langle\langle x_{m-1}\rangle\rangle\psi_w$ and $\varphi_c \triangleq [[y_0]] \dots [[y_{n-1}]]\psi_c$ two SLK sentences. Then:

- Memoryless uniform shared strategies $f_{w0}, \dots, f_{w(m-1)}$ are *witness strategies* for φ_w at g iff *(i)* $f_{wi} \in \mathit{UStr}_{\mathsf{sharing}(\psi_w, x_i)}$ for all $0 \le i < m$ and *(ii)* $\mathcal{I}, \chi_w, g \models_{\text{SLK}} \psi_w$ where $\chi_w \triangleq \{(x_i, f_{wi}) \mid 0 \le i < m\}$.
- Memoryless uniform shared strategies $f_{c0}, \dots, f_{c(n-1)}$ are *counterexample strategies* for φ_c at g iff *(i)* $f_{ci} \in \mathit{UStr}_{\mathsf{sharing}(\psi_c, y_i)}$ for all $0 \le i < n$ and *(ii)* $\mathcal{I}, \chi_c, g \not\models_{\text{SLK}} \psi_c$ where $\chi_c \triangleq \{(y_i, f_{ci}) \mid 0 \le i < n\}$.

Consider the toy model in Figure 3.1 and the SLK formulas $\varphi_1 \triangleq \langle\langle e\rangle\rangle\langle\langle x\rangle\rangle\langle\langle y\rangle\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{X}\, p_1$ and $\varphi_2 \triangleq [[e]][[x]][[y]](\mathrm{E}, e)(1, x)(2, y)\mathsf{X}\, p_2$. The formula φ_1 means that *"there exist strategies for all the agents, such that player 1 wins in the next round"*. The formula φ_2 expresses that *"for all strategies of the agents, player 2 wins in the next round"*. By SLK semantics, we have $\mathcal{I}, \emptyset, g_{\mathrm{g}} \models_{\text{SLK}} \varphi_1$ and $\mathcal{I}, \emptyset, g_{\mathrm{g}} \not\models_{\text{SLK}} \varphi_2$. Incidentally, the following strategies are both *witness strategies* for φ_1 and *counterexample strategies* for φ_2 at g_g:

$$\begin{array}{lll} f_e(g_\mathrm{g}) \triangleq \mathrm{i} & f_x(g_\mathrm{g}) \triangleq \mathrm{p} & f_y(g_\mathrm{g}) \triangleq \mathrm{r} \\ f_e(g_1) \triangleq \mathrm{i} & f_x(g_1) \triangleq \mathrm{i} & f_y(g_1) \triangleq \mathrm{i} \\ f_e(g_2) \triangleq \mathrm{i} & f_x(g_2) \triangleq \mathrm{i} & f_y(g_2) \triangleq \mathrm{i} \end{array}$$

The two concepts are *duals* of each other in the sense that if f is a witness strategy for $\langle\!\langle x \rangle\!\rangle\psi$ at g, then it is a counterexample strategy for $[\![x]\!]\neg\psi$ at g (and vice versa).

Lemma 4.3. Let $\mathcal{I}$ be an interpreted system, $g \in G$ a global state, ψ an agent-closed SLK formula with $\mathsf{free}(\psi) = \{x_0, \ldots, x_{n-1}\}$, and $f_0, \ldots, f_{n-1}$ memoryless uniform shared strategies such that $f_i \in \mathit{UStr}_{\mathsf{sharing}(x_i,\psi)}$ for $0 \le i < n$. Then $f_0, \ldots, f_{n-1}$ are witness strategies of $\langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi$ at g iff $f_0, \ldots, f_{n-1}$ are counterexample strategies of $[\![x_0]\!] \ldots [\![x_{n-1}]\!]\neg\psi$ at g.

Proof. We prove both directions of the equivalence separately:

$\Rightarrow$: Assume that $f_0, \ldots, f_{n-1}$ are witness strategies of $\langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi$ at g. By Definition 4.13, we have $f_i \in \mathit{UStr}_{\mathsf{sharing}(\psi,x_i)}$ for all $0 \le i < n$ and $\mathcal{I}, \chi, g \models \psi$ where $\chi \triangleq \{(x_i, f_i) \mid 0 \le i < n\}$. By Definition 2.22, we have $\mathsf{sharing}(\neg\varphi, x) = \mathsf{sharing}(\varphi, x)$ for all SLK formulas $\varphi \in \mathit{SLK}$ and variables $x \in \mathit{Var}$ so $f_i \in \mathit{UStr}_{\mathsf{sharing}(\neg\psi,x_i)}$ for all $0 \le i < n$. Since $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \psi$, $\mathcal{I}, \chi, g \not\models_{\mathrm{SLK}} \neg\psi$ holds by SLK semantics (Definition 4.3). Hence, by Definition 4.13, $f_0, \ldots, f_{n-1}$ are counterexample strategies for $[\![x_0]\!] \ldots [\![x_{n-1}]\!]\neg\psi$ at g.

$\Leftarrow$: Assume that $f_0, \ldots, f_{n-1}$ are counterexample strategies of $[\![x_0]\!] \ldots [\![x_{n-1}]\!]\neg\psi$ at g. By Definition 4.13, we have $f_i \in \mathit{UStr}_{\mathsf{sharing}(\neg\psi,x_i)}$ for all $0 \le i < n$ and $\mathcal{I}, \chi, g \not\models_{\mathrm{SLK}} \neg\psi$ where $\chi \triangleq \{(x_i, f_i) \mid 0 \le i < n\}$. Again, we have $f_i \in \mathit{UStr}_{\mathsf{sharing}(\psi,x_i)}$ for all $0 \le i < n$ by Definition 2.22. Since $\mathcal{I}, \chi, g \not\models_{\mathrm{SLK}} \neg\psi$, $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \psi$ holds by SLK semantics (Definition 4.3). Hence, by Definition 4.13, $f_0, \ldots, f_{n-1}$ are witness strategies of $\langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi$ at g.

□

This duality will allow us to focus on witness strategies and their synthesis only: Suppose an SLK sentence $\varphi \triangleq [\![x_0]\!] \ldots [\![x_{n-1}]\!]\psi$ does not hold hold at a state $g \in G$ in an interpreted system $\mathcal{I}$ and we want to construct counterexample strategies for φ at g. Since $\mathcal{I}, \emptyset, g \not\models_{\mathrm{SLK}} \varphi$, we must have $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \neg\varphi$ by SLK semantics (see Definition 4.3). Using the equivalence $\neg [\![x_0]\!] \ldots [\![x_{n-1}]\!]\psi \equiv \langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\neg\psi$, we get $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\neg\psi$. Finally, we construct witness strategies for $\neg\varphi$ at g, which are also counterexample strategies for φ at g by Lemma 4.3.

The procedure for constructing a counterexample strategy relies on witness strategy synthesis. Before describing how witness strategies can be retrieved, we need to show that they always exist when an SLK formula holds.

Lemma 4.4. Let $\mathcal{I}$ be an interpreted system, ψ an agent-closed SLK formula such that $\mathsf{free}(\psi) = \{x_0, \ldots, x_{n-1}\}$, and $\varphi \triangleq \langle\!\langle x_0 \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi$ an SLK sentence. Then the following holds: $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \varphi$ iff there exist witness strategies $f_0, \ldots, f_{n-1}$ for φ at g.

Proof. We prove both directions of the equivalence separately:

$\Rightarrow$: Assume that $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \varphi$. By SLK semantics (Definition 4.3), there exist strategies $f_0, \ldots, f_{n-1}$ such that $f_i \in \mathit{UStr}_{\mathsf{sharing}(\langle\!\langle x_{i+1} \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi,x_i)}$ for all $0 \le i < n$ and $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \psi$ where $\chi = \{(x_i, f_i) \mid 0 \le i < n\}$. By Definition 2.22, we have $\mathsf{sharing}(\langle\!\langle x_{i+1} \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi, x_i) = \mathsf{sharing}(\psi, x_i)$ for all $0 \le i < n$. Hence $f_0, \ldots, f_{n-1}$ satisfy both conditions for being witness strategies of φ at g (see Definition 4.13).

$\Leftarrow$: Assume that there exist witness strategies $f_0, \ldots, f_{n-1}$ for φ at g. By Definition 4.13, we have $f_i \in \mathit{UStr}_{\mathsf{sharing}(\psi,x_i)}$ for all $0 \le i < n$ and $\mathcal{I}, \chi, g \models_{\mathrm{SLK}} \psi$ where $\chi \triangleq \{(x_i, f_i) \mid 0. \le i < n\}$. By Definition 2.22, we have $\mathsf{sharing}(\psi, x_i) = \mathsf{sharing}(\langle\!\langle x_{i+1} \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi, x_i)$. Hence, we get $f_i \in \mathit{UStr}_{\mathsf{sharing}(\langle\!\langle x_{i+1} \rangle\!\rangle \ldots \langle\!\langle x_{n-1} \rangle\!\rangle\psi,x_i)}$ for all $0 \le i < n$. Therefore, $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \varphi$ by SLK semantics (Definition 4.3).

□

It remains to explain how witness strategies can be synthesised using the Slk model checking algorithm we introduced in Subsection 4.2.2. Consider an Slk sentence $\varphi \triangleq \langle\langle x_0 \rangle\rangle \dots \langle\langle x_{n-1} \rangle\rangle\, \psi$ that holds at a global state $g \in G$ in some interpreted system $\mathcal{I}$. Since $\mathcal{I}, \emptyset, g \models_{\text{SLK}} \varphi$, by Lemma 4.4, there must be some witness strategies $f_0, \dots, f_{n-1}$ for φ at g, which we want to synthesise. The corresponding assignment $\chi \in \mathit{Asg}$ on the variables $x_0, \dots, x_{n-1}$ satisfies $\mathcal{I}, \chi, g \models_{\text{SLK}} \psi$. Let $E \triangleq \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$ be the set of extended states which guarantee ψ in $\mathcal{I}$ under the empty binding. E contains all possible extended states $\langle g', v' \rangle$ such that $\mathcal{I}, g', v' \models_{\text{SLK}} \psi$. Hence, it must be the case that $\langle g, v \rangle \in E$ where v is some variable assignment which extends χ.

Therefore, in order to synthesise witness strategies for φ at g, it suffices to find an extended state $\langle g, v \rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$ such that the strategies $v(x_0), \dots, v(x_{n-1})$ are uniform with respect to the agents which share them. The witness strategies for φ at g are then $v(x_0), \dots, v(x_{n-1})$.

Lemma 4.5. Let $\mathcal{I}$ be an interpreted system, ψ an agent-closed Slk formula such that $\mathsf{free}(\psi) = \{x_0, \dots, x_{n-1}\}$, and $\varphi \triangleq \langle\langle x_0 \rangle\rangle \dots \langle\langle x_{n-1} \rangle\rangle \psi$ an Slk sentence. Then the following properties hold:

1. For all variable assignments $\langle g, v \rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$ such that $v(x_i) \in \mathit{UStr}_{\mathsf{sharing}(\psi, x_i)}$ for $0 \leq i < n$, $v(x_0), \dots, v(x_{n-1})$ are witness strategies for φ at g.

2. If there exist witness strategies for φ at g, then there exists a variable assignment $v \in \mathit{VAsg}$ such that $v(x_i) = f_i$ for $0 \leq i < n$ and $\langle g, v \rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$.

Proof. We prove both properties separately:

1. Take an arbitrary extended state $\langle g, v \rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$ such that $v(x_i) \in \mathit{UStr}_{\mathsf{sharing}(\psi, x_i)}$ for $0 \leq i < n$. As we have explained in the proof of Theorem 4.2, $\langle g, v \rangle$ guarantees ψ in $\mathcal{I}$ under $\emptyset$. By Definition 4.9, $\mathcal{I}, g, v \models_{\text{SLK}} \psi$. By Slk semantics (Definition 4.3), we have $\mathcal{I}, \chi, g \models_{\text{SLK}} \psi$ where $\chi \triangleq \{(x_i, v(x_i)) \mid 0 \leq i < n\}$ since $\mathsf{free}(\psi) \subseteq \{x_0, \dots, x_{n-1}\} \subseteq \mathrm{dom}(v)$. Hence, $v(x_0), \dots, v(x_{m-1})$ satisfy both conditions for being witness strategies for φ at g (see Definition 4.13).

2. Assume that $f_0, \dots, f_{n-1}$ are witness strategies for φ at g. By Definition 4.13, $f_i \in \mathit{UStr}_{\mathsf{sharing}(\psi, x_i)}$ for all $0 \leq i < n$ and $\mathcal{I}, \chi, g \models_{\text{SLK}} \psi$ where $\chi \triangleq \{(x_i, f_i) \mid 0 \leq i < n\}$. Since $\mathrm{dom}(\chi) = \mathsf{free}(\psi)$, by Slk semantics (Definition 4.3), we have $\mathcal{I}, g, v \models_{\text{SLK}} \psi$ for all $v \in \mathit{VAsg}$ such that $\chi \subseteq v$. There must exist at least one such variable assignment v because $\mathrm{dom}(\chi) \subseteq \mathit{Var}$ (simply set $v(x) \triangleq \chi(x)$ for $x \in \mathrm{dom}(\chi)$ and assign arbitrary strategies to variables $y \in \mathit{Var} \setminus \mathrm{dom}(\chi)$). By Definition 4.9, $\langle g, v \rangle \in \mathit{Ext}$ guarantees ψ in $\mathcal{I}$ under $\emptyset$. Therefore, $\langle g, v \rangle \in \mathsf{SAT}^{\mathcal{I}}_{\text{SLK}}(\psi, \emptyset)$ (see proof of Theorem 4.2).

□

Informally, the first property in Lemma 4.5 expresses *soundness* of the approach, i.e. that it will return only witness strategies for φ at g. Conversely, the second property asserts *completeness* of the approach, i.e. that it will return witness strategies for φ at g, if they exist.

4.2.4 Symbolic Implementation

In this subsection, we discuss how the algorithm presented in Subsection 4.2.2 can be implemented symbolically using BDDs. As explained in Subsection 2.3.2, BDDs are a very efficient representation for manipulation of Boolean formulas. BDDs are used by many existing model checkers including MCMAS (see Subsection 2.5.2), which uses them for model checking Ctl and Atl [65]. We present here a modification of the symbolic implementation for Slk.

We start by representing the parameters of the interpreted system by means of Boolean formulas [81]. Given an interpreted system:

$$\mathcal{I} = \left\langle (L_i, \mathit{Act}_i, P_i, t_i)_{i \in \mathit{Agt}}, I, h \right\rangle$$

we can represent global states and joint actions as follows [65, 81]:

- For every agent $i \in \mathit{Agt}$, we can encode its set of internal states L_i with $\mathsf{nv}(i) = \lceil \log_2 |L_i| \rceil$ Boolean variables. Thus, a *global state* $g \in G$ can be encoded as a Boolean vector $\overline{v} = (v_0, \dots, v_{N-1})$, where $N = \sum_{i \in \mathit{Agt}} \mathsf{nv}(i)$.

- For every agent $i \in Agt$, we can encode its set of actions Act_i with $\mathsf{na}(i) = \lceil \log_2 |Act_i| \rceil$ Boolean variables. Thus, a *joint action* $a \in Act$ can be encoded as a Boolean vector $\overline{w} = (w_0, \dots, w_{M-1})$, where $M = \sum_{i \in Agt} \mathsf{na}(i)$.

An example of how this encoding can be done is provided in Subsection 3.3.2.

Encoding of Extended States

In order to implement our new SLK model checking algorithm, we need to represent sets of *extended states* (see Definition 4.8), which consist of a global state and a variable assignment. We thus represent the variable assignment explicitly using Boolean variables as well.

Since the number of strategy variables (and hence the domain of a variable assignment) depends on the SLK formula we are checking, the total number of Boolean variables also depends on the formula. Let $\varphi \in SLK$ be an SLK sentence and $\mathsf{vars}(\varphi) \subseteq Var$ the set of variables quantified in φ, e.g. $\mathsf{vars}(\langle\langle e \rangle\rangle [\![x]\!] \langle\langle y \rangle\rangle (\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\, [\neg p_1 \wedge \neg p_2 \wedge \langle\langle z \rangle\rangle (2, z)\mathsf{X}\, p_2]) = \{e, x, y, z\}$. For each variable $x \in \mathsf{vars}(\varphi)$, we represent the strategy associated with x using a number of Boolean variables.

Let us now consider an arbitrary variable $x \in \mathsf{vars}(\varphi)$. In the SLK model checking algorithm (Definition 4.12), the variable quantifies over strategies $f \in UStr_{\mathsf{sharing}(\varphi, x)}$. The domain and the range of a strategy $f \in UStr_A$ are G and Act_A respectively (see Definition 4.2). Hence, we could easily represent it using $|G| \times \lceil \log_2 |Act_A| \rceil$ Boolean variables, i.e. we would encode the action associated with each global state. For example, the following strategy for player 1 ($A = \{1\}$) in the toy model (see Figure 3.1):

$$f(g_{\mathrm{g}}) \triangleq \mathrm{r} \qquad\qquad f(g_1) \triangleq \mathrm{i} \qquad\qquad f(g_2) \triangleq \mathrm{i}$$

could be encoded using $3 \times \lceil \log_2 |\mathrm{r}, \mathrm{p}, \mathrm{s}, \mathrm{i}| \rceil = 6$ Boolean variables:

$$\underbrace{\neg b_0 \wedge \neg b_1}_{f(g_{\mathrm{g}})=\mathrm{r}} \wedge \underbrace{b_2 \wedge b_3}_{f(g_1)=\mathrm{i}} \wedge \underbrace{b_4 \wedge b_5}_{f(g_2)=\mathrm{i}}$$

We can see that this representation is not the most compact one. Namely, we do not need 2 Boolean variables to encode the action in state g_1 because $P_1(s, s_1) = \{\mathrm{i}\}$. In fact, we do not need any Boolean variables to encode $f(g_1) = f(g_2) = \mathrm{i}$ because i is the only available action in both g_1 and g_2 so agent 1 always has to perform it in these states. Hence, a more compact representation can be obtained by using only $\sum_{g \in G} \lceil \log_2 |\bigcap_{i \in A} P_i(l_{i\mathrm{E}}(g))| \rceil$ Boolean variables. The strategy f for the toy model would now be encoded using $\lceil \log_2 |\{\mathrm{r}, \mathrm{p}, \mathrm{s}\}| \rceil + 2 \lceil \log_2 |\{\mathrm{i}\}| \rceil = 2$ Boolean variables as $\neg b_0 \wedge \neg b_1$, which is optimal since there are exactly three possible strategies available to agent 1. More generally, this encoding is almost optimal[7] for shared memoryless strategies.

However, our strategies are also *uniform*. If have a strategy $f \in UStr_{\{i\}}$ for an agent $i \in Agt$, then for all states $g_1, g_2 \in G$, $g_1 \sim_i g_2$ implies $f(g_1) = f(g_2)$. Hence, there is no point in storing both actions $f(g_1)$ and $f(g_2)$. Consider some interpreted system with 2 agents a, b and 3 reachable global states g_1, g_2, g_3 where $g_1 \sim_a g_2$ and $g_2 \sim_b g_3$. Furthermore, assume the following protocols:

$$P_a(l_{a\mathrm{E}}(g_1)) = P_a(l_{a\mathrm{E}}(g_2)) \triangleq \{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3\} \qquad\qquad P_b(l_{b\mathrm{E}}(g_1)) \triangleq \{\mathrm{a}_1, \mathrm{a}_2\}$$
$$P_a(l_{a\mathrm{E}}(g_3)) \triangleq \{\mathrm{a}_2, \mathrm{a}_3\} \qquad\qquad P_b(l_{b\mathrm{E}}(g_2)) = P_b(l_{b\mathrm{E}}(g_3)) \triangleq \{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3\}$$

How many Boolean variables do we need to encode a strategy $f \in UStr_{\{a,b\}}$ for both agents? Let us forget for a while that f is uniform. The more compact representation we found would require $\sum_{g \in G} \lceil \log_2 |P_a(l_{a\mathrm{E}}(g)) \cap P_b(l_{b\mathrm{E}}(g)))| \rceil = \lceil \log_2 |\{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3\}| \rceil + \lceil \log_2 |\{\mathrm{a}_1, \mathrm{a}_2\}| \rceil + \lceil \log_2 |\{\mathrm{a}_2, \mathrm{a}_3\}| \rceil = 4$ Boolean variables. What are the possible strategies f? We require $f(g) \in P_i(l_{i\mathrm{E}}(g))$ for all agents $i \in \{a, b\}$ and global states $g \in G$: $f(g_1) \in \{\mathrm{a}_1, \mathrm{a}_2\}$, $f(g_2) \in \{\mathrm{a}_1, \mathrm{a}_2, \mathrm{a}_3\}$, and $f(g_2) \in \{\mathrm{a}_2, \mathrm{a}_3\}$. So there are 12 possible shared memoryless strategies. But f must be *uniform*. Since $g_1 \sim_a g_2$, and $g_2 \sim_b g_3$, we must have $f(g_1) = f(g_2) = f(g_3)$. Therefore, there is only one possible uniform strategy defined as $f(g) \triangleq \mathrm{a}_2$ for all $g \in G$. This answers our first question: We do not need any Boolean variables.

[7]It is optimal as long as we store each action individually. Consider the situation when an agent can perform 3 actions in state s_1 and 5 actions in s_2. Our encoding will require $\lceil \log_2 3 \rceil + \lceil \log_2 5 \rceil = 5$ Boolean variables to represent a strategy. Since there are $3 \times 5 = 15$ possible strategies, it is possible to encode a strategy using only $\lceil \log_2 15 \rceil = 4$ Boolean variables.

The example above showed us that for a given set of agents $A \subseteq Agt$, the epistemic accessibility relations $\sim_i$ with $i \in A$ induce regions of the global state space, to which the uniform strategies $f \in \mathit{UStr}_A$ must assign the same action. It turns out that these regions are equivalence classes with respect to the common epistemic accessibility relation $\sim^{\mathsf{C}}_A$. We shall now prove this statement.

Lemma 4.6. Let $\mathcal{I}$ be an interpreted system and $A \subseteq Agt$ a set of agents. Then a memoryless strategy $f : G \to Act_A$ is *uniform* iff for each set of global states in the quotient set $S \in G/{\sim^{\mathsf{C}}_A}$, we have $f(s_1) = f(s_2)$ for all $s_1, s_2 \in S$.

Proof. We shall prove both directions of the equivalence separately.

⇒: Assume that f is uniform and take an arbitrary set $S \in G/{\sim^{\mathsf{C}}_A}$. Furthermore, take arbitrary global states $g_1, g_2 \in S$. By the definition of quotient set, we have $g_1 \sim^{\mathsf{C}}_A g_2$. There are two cases:

- $g_1 = g_2$. Trivially, $f(g_1) = f(g_2)$.
- By the definition of common epistemic accessibility relation (Definition 2.31), there is a chain of global states $g'_1, g'_2, \ldots, g'_n \in G$ and agents $i_1, i_2, \ldots, i_{n+1} \in A$ with $n \geq 0$ such that $g_1 \sim_{i_1} g'_1 \sim_{i_2} g'_2 \ldots g'_n \sim_{i_{n+1}} g_2$. By uniformity of f (Definition 4.2), we get $f(g_1) = f(g'_1) = \cdots = f(g'_n) = f(g_2)$.

⇐: Assume that for each $S \in G/{\sim^{\mathsf{C}}_A}$, we have $f(s_1) = f(s_2)$ for all $s_1, s_2 \in S$. Take an arbitrary agent $i \in A$ and global states $s_1, s_2 \in G$ such that $s_1 \sim_i s_2$. To prove uniformity of f, we need to show that $f(s_1) = f(s_2)$. Since $s_1 \sim_i s_2$, we also have $s_1 \sim^{\mathsf{C}}_A s_2$. Hence, s_2 belongs to the equivalence class $[s_1]_{\sim^{\mathsf{C}}_A}$. By definition of an equivalence class, it must be the case that $s_1 \in [s_1]_{\sim^{\mathsf{C}}_A}$ and $[s_1]_{\sim^{\mathsf{C}}_A} \in G/\sim^{\mathsf{C}}_A$. Since $s_1, s_2 \in [s_1]_{\sim^{\mathsf{C}}_A}$ and $[s_1]_{\sim^{\mathsf{C}}_A} \in G/{\sim^{\mathsf{C}}_A}$, we get $f(s_1) = f(s_2)$ by the initial assumption as required.

□

This allows us to present an even more compact representation of a strategy $f \in \mathit{UStr}_A$ with $A \subseteq Agt$ in an interpreted system $\mathcal{I}$. We only need to store one action for each *shared local state* $S \in G/{\sim^{\mathsf{C}}_A}$. Thus, we can represent f using $\sum_{S \in G/\sim^{\mathsf{C}}_A} \left\lceil \log_2 \left| \bigcap_{g \in S} \bigcap_{i \in A} P_i(l_{i\mathrm{E}}(g)) \right| \right\rceil$ Boolean variables. Finally, a variable assignment $v \in \mathit{VAsg}$ for a formula $\varphi \in \mathit{SLK}$ can be represented using a Boolean vector $\overline{u} = (u_0, \ldots, u_{K-1})$ such that:

$$K = \sum_{x \in \mathsf{vars}(\varphi)} \sum_{S \in G_x} \left\lceil \log_2 \left| \bigcap_{g \in S} \bigcap_{i \in \mathsf{sharing}(\varphi, x)} P_i(l_{i\mathrm{E}}(g)) \right| \right\rceil$$

where $G_x \triangleq G/ \sim^{\mathsf{C}}_{\mathsf{sharing}(\varphi,x)}$ is the set of shared local states for variable x. Note that despite both optimisations, the worst case still remains $K = |\mathsf{vars}(\varphi)| \times |G| \times \lceil \log_2 (\max_{i \in Agt} |Act_i|) \rceil$, i.e. we need polynomially many Boolean variables with respect to both the size of the model $|\mathcal{I}|$ and the number of strategy variables in the formula $|\mathsf{vars}(\varphi)|$.

An extended state $\langle g, v \rangle \in \mathit{Ext}$ can be represented by a concatenation of the Boolean vectors $\overline{v}_g$ and $\overline{u}_v$, which can in turn be identified with Boolean formulas, represented by conjunctions of literals (as in Subsection 2.3.2). A set of extended states $E \subseteq \mathit{Ext}$ can be expressed as the disjunction of the Boolean formulas encoding each extended state in E.

Encoding of the Algorithm

Given a binding $b \in \mathit{Bnd}$ such that $\mathrm{dom}(b) = Agt$, we define a formula $S^b(e, a)$, where $e \in \mathit{Ext}$ and $a \in Act$, representing the *strategy restrictions* of the implied transition relation (see Definition 4.10). The formula asserts that all agents act according to their strategies:

$$S^b(\langle g, v \rangle, a) \triangleq \forall i \in Agt.\, v(b(i))(g) = a_i(a)$$

Let $\overline{v}$, $\overline{v'}$, $\overline{w}$, and $\overline{u}$ be the Boolean vectors for representing current global states, next global states, joint actions, and variable assignments respectively. We can encode strategy restrictions as a Boolean formula $S^b(\overline{v}, \overline{w}, \overline{u})$:

$$S^b(\overline{v}, \overline{w}, \overline{u}) = \bigwedge_{i \in Agt} \bigvee_{l \in L_{i\mathrm{E}}} \left[l(\overline{v}_{i\mathrm{E}}) \wedge \bigvee_{a \in P_i(l)} a(\overline{w}_{i\mathrm{E}}) \wedge a(\overline{u}_{v(b(i)),l}) \right]$$

where:

- $l(\overline{v}_{i\mathrm{E}})$ is the Boolean formula representing that the local state of agent $i \in Agt$ is $l \in L_{i\mathrm{E}}$;
- $a(\overline{w}_{i\mathrm{E}})$ is the Boolean formula representing that the action of agent $i \in Agt$ is $a \in Act_i$;
- $a(\overline{u}_{v(x),l})$ is the Boolean formula representing the fact that the action in local state[8] l assigned by the strategy mapped to variable x is a.

We can represent the global protocol and evolution function as Boolean formulas $P(\overline{v}, \overline{w})$ and $t(\overline{v}, \overline{w}, \overline{v'})$ by taking the conjunctions of Boolean formulas representing the individual agent's protocols P_i and evolution functions t_i ($i \in Agt$), like we did in Subsection 3.3.2. The Boolean formula $R^b_t(\overline{v}, \overline{w}, \overline{v'})$ for the *implied transition relation* $\rightarrow^b_v \subseteq G \times G$ (see Definition 4.10) is then constructed from the conjunction of the Boolean formulas representing the global protocol, global evolution function, and strategy restrictions:

$$R^b_t(\overline{v}, \overline{v'}, \overline{u}) = \exists \overline{w}.\, P(\overline{v}, \overline{w}) \wedge t(\overline{v}, \overline{w}, \overline{v'}) \wedge S^b(\overline{v}, \overline{w}, \overline{u})$$

Note that we quantify over actions, encoded as $\overline{w}$, as in Subsection 3.3.2, but we keep the variable assignment in the extra parameter[9] $\overline{u}$. Quantification over the variable assignment is performed when a strategy quantifier ($\langle\!\langle x \rangle\!\rangle$, $[\![x]\!]$) is encountered. Also note that the strategy restrictions S^b depend on the binding b and thus have to be recomputed when the binding is updated, i.e. when the agent binding operator (i, x) with $i \in Agt$ and $x \in Var$ is encountered. This is not the case for the other Boolean formulas ($P(\overline{v}, \overline{w})$ and $t(\overline{v}, \overline{w}, \overline{v'})$), which are constant for a given interpreted system.

The *individual epistemic accessibility relation* $R^{\mathsf{K}}_i(\overline{v}, \overline{v'})$ for each agent $i \in Agt$ and the other epistemic accessibility relations are encoded in a similar way [65]. The computation of the set of *reachable states* G is explained at the end of Subsection 3.3.2.

Finally, the algorithm $\mathsf{SAT}_{\mathrm{SLK}} : SLK \times Bnd \rightarrow 2^{Ext}$ can be translated into operations on BDDs representing encoded sets of extended states.

Strategy Synthesis

Synthesising *witness* and *counterexample strategies* (see Definition 4.13) using the symbolic implementation is very simple because our representation of variable assignments ensures that all strategies are uniform (see Definition 4.2).

Assume that we want to synthesise witness strategies for an SLK sentence $\varphi \triangleq \langle\!\langle x_0 \rangle\!\rangle \dots \langle\!\langle x_{n-1} \rangle\!\rangle \psi$ at a global state $g \in G$. Since $\mathcal{I}, \emptyset, g \models_{\mathrm{SLK}} \varphi$, there exist witness strategies for φ at g (see Lemma 4.4). We calculate $E = \mathsf{SAT}^{\mathcal{I}}_{\mathrm{SLK}}(\psi, \emptyset)$ and pick an arbitrary extended state $\langle g, v \rangle \in E$. Note that all strategies in v are uniform due to our representation. By Lemma 4.5, there exists at least one such extended state $\langle g, v \rangle$ (because there exist witness strategies for φ at g) and for all such extended states, $v(x_0), \dots, v(x_{n-1})$ are witness strategies for φ at g. Witness strategy synthesis for φ at g can be implemented symbolically as follows:

1. Calculate $E := \mathsf{SAT}^{\mathcal{I}}_{\mathrm{SLK}}(\psi)$. This is performed using the symbolic implementation presented earlier in this subsection.
2. Filter out extended states with a different global state $E' := \{(g', v') \in E \mid g' = g\}$. The symbolic representation of this operation is $E'(\overline{v}, \overline{u}) := E(\overline{v}, \overline{u}) \wedge g(\overline{v})$.

[8]Strictly speaking, the strategy maps *global* states to actions. However, as explained earlier, there exists a set $S \subseteq G/\sim^{\mathsf{C}}_{\mathsf{sharing}(\varphi,x)}$ such that, for all global states $g \in G$, if $l_{i\mathrm{E}}(g) = l$ then $g \in S$. Thus, we can also interpret the strategy as a mapping from *local* states to actions.

[9]Observe that we do not need to allocate BDD variables $\overline{u'}$ for the next value of the variable assignment because it does not change in a temporal transition (see Definition 4.11).

3. Pick an arbitrary extended state $\langle g, v\rangle \in \mathit{Ext}$. This is equivalent to selecting one conjunct (also referred to as *minterm*) $C(\overline{v}, \overline{u})$ from $E'(\overline{v}, \overline{u})$. BDD packages usually provide a built-in function for this operation[10].

4. The conjunct $C(\overline{v}, \overline{u})$ encodes the extended state $\langle g, v\rangle$, where v contains the witness strategies. If we want to find the next action $f_i(g')$ of a strategy $f_i = v(x_i)$ for $0 \leq i < n$ at a global state $g' \in G$, we iterate over all possible actions $a \in \bigcap_{j \in \mathsf{sharing}(\psi, x_i)} P_j(l_{j\mathrm{E}}(g'))$ and pick one such that $C(\overline{v}, \overline{u}) \wedge a(\overline{u}_{v(x_i),g'})$ is not equivalent to falsity, where $a(\overline{u}_{v(x_i),g'})$ is the Boolean formula representing the fact that the next action of the strategy mapped to variable x_i at the global state g' is a.

Complexity

Having explained how the Slk model checking algorithm can be implemented symbolically, it is natural to ask what its complexity is. As we have just shown, the symbolic encoding requires polynomially many Boolean variables. Therefore, the algorithm runs in *exponential time* with respect to both the size of the model $|\mathcal{I}|$ and the number of quantified variables $|\mathsf{vars}(\varphi)|$.

Theorem 4.3. Let $\mathcal{I} = \left\langle (L_i, Act_i, P_i, t_i)_{i \in Agt}, I, h \right\rangle$ be an arbitrary interpreted and $\varphi \in \mathit{SLK}$ be an Slk sentence. The worst case time complexity of the symbolic implementation is:

$$2^{O(|G| \times |\mathsf{vars}(\varphi)| \times \log_2 |Act|)}$$

where G is the set of reachable global states of $\mathcal{I}$.

Proof (Sketch). The symbolic implementation uses:

- $O(\log_2 |G|)$ Boolean variables to represent the *current global state*;
- $O(\log_2 |G|)$ Boolean variables to represent the *next global state*;
- $O(\log_2 |Act|)$ Boolean variables to represent the *joint actions*;
- $O(|G| \times |\mathsf{vars}(\varphi)| \times \log_2 (\max_{i \in Agt} |Act_i|))$ Boolean variables to represent the *variable assignment.*

The total number of Boolean variables needed is thus $O(|G| \times |\mathsf{vars}(\varphi)| \times \log_2 |Act|)$. Since BDD operations take polynomial time with respect to the size of the relevant BDDs in the worst case [49], they take at most exponential[11] time in the number of Boolean variables. Hence, our claim follows. □

The proof shows that the complexity of the algorithm is dominated by the encoding of the variable assignments. One of the main future directions in Slk verification is thus finding more compact representations of extended states.

Observe that the symbolic algorithm can use more than a polynomial amount of space because it performs operations on sets of extended states. While the procedure described in Subsection 4.2.1 uses only a polynomial amount of space (as it operates on individual states and assignments), we believe it is unlikely that such an explicit approach would outperform the symbolic algorithm.

4.3 Summary

In this chapter, we introduced a novel fragment of Sl called Epistemic Strategy Logic (Slk), which is defined on imperfect recall semantics with incomplete knowledge (i.e. agents have no memory of the past and do not have complete knowledge of the global state). Slk combines Ltl temporal operators, Sl strategy quantifiers, and epistemic modalities. The result is a formalism which is even more expressive than Sl because it can express combined concepts such as knowledge about Nash equilibria. We showed that, despite its expressive power, the Slk model checking problem belongs to the PSpace complexity

[10]If no such function is available, we can skip step 3 ($C(\overline{v}, \overline{u}) := E'(\overline{v}, \overline{u})$) and refine the conjunct upon each lookup in step 4 ($C(\overline{v}, \overline{u}) := C(\overline{v}, \overline{u}) \wedge a(\overline{u}_{v(x_i),g'})$).

[11]Given v Boolean variables, a BDD depending only on these variables can have at most 2^v nodes.

class. Furthermore, we provided an exponential-time model checking algorithm for Slk which admits an efficient symbolic implementation and supports witness/counterexample strategy synthesis. Hence, the algorithm can be used to automatically synthesise agents' behaviour which ensures an arbitrary Slk specification.

The first part of the chapter focused on the basic Slk *theory*. We gave the syntax and semantics of the logic, which is based on *uniform* strategies where the next actions assigned to two global states indistinguishable by an agent must be the same. This property ensures that the strategies are executable by the agents despite incomplete information. We then compared the logic to the original Sl and discussed its limitations due to memoryless and non-behavioural strategies.

The second part of the chapter discussed Slk *model checking*. We started by defining the model checking problem formally and proved that it belongs to the PSpace complexity class with respect to both the size of the model and the formula. We then provided a practical model checking algorithm for Slk based on the concept of extended states and showed that it is correct. Furthermore, we showed how it can be used to synthesise witness and counterexample strategies for Slk sentences of the form $\langle\!\langle x_0 \rangle\!\rangle \cdots \langle\!\langle x_{m-1} \rangle\!\rangle \psi$ and $[\![y_0]\!] \cdots [\![y_{n-1}]\!] \psi$ respectively. Finally, we described how the algorithm can be implemented symbolically and showed that it has worst-case exponential time complexity.

In Section 6.2, we will describe how we developed an extension of MCMAS which implements the Slk model checking algorithm introduced in this chapter using BDDs.

Chapter 5

One-Goal Strategy Logic

One-Goal Strategy Logic (SL[1G]) is another syntactic fragment of SL (see Subsection 2.2.5), which was introduced in [72]. Unlike SLK, SL[1G] is defined on perfect recall semantics with complete information (i.e. agents have perfect memory of the past and complete knowledge of the global state). We provide the first practical model checking algorithm for SL[1G]. We prove its correctness and show that it has optimal worst-case time complexity. We explain how it can be extended to support general strategy synthesis and provide an efficient symbolic implementation. This chapter is split into two parts: Section 5.1 constitutes a very short revision of SL[1G] and Section 5.2 describes the new model checking algorithm we have developed.

5.1 Logic

As we have already mentioned in Section 3.2, SL[1G] is one of the syntactic fragments of full SL introduced in [72]. Since it is the least expressive fragment, it does not have the full power of SL and *cannot* express game-theoretic concepts like Nash equilibria. On the other hand, the "rewards" for this reduction in expressiveness are some very desirable properties [73]:

1. **Behavioural strategies.** Informally, SL[1G] strategies depend only on the history of the game and the next actions of other agents in the current state. More importantly they do *not* depend on other agents' actions in the future or in other counterfactual games. While this might not seem very important, it is a fundamental property on which our model checking algorithm relies.

2. 2EXPTIME-COMPLETE **model checking complexity.** While this is still a very high complexity, it is a massive improvement over the NONELEMENTARYSPACE-HARD complexity of full SL. More importantly, SL[1G] has exactly the *same complexity* as ATL* while being *strictly more expressive* [72]. Therefore, our new algorithm presented in Section 5.2 is also optimal[1] for model checking ATL*.

   Note that we are referring to the complexity with respect to the size of the *formula*. The complexity of ATL* and all perfect recall fragments of SL with respect to the size of the *model* is P. Recall that SLK, which has imperfect recall, is in PSPACE with respect to *both* the size of the model and the formula (see Theorem 4.1).

3. **Decidable satisfiability.** While this property is not important for our purposes, it is still a desirable result. Note that SL[1G] is the only fragment of SL for which it is known that satisfiability is decidable (see Section 3.2 for an overview of the fragments and their properties).

Recall that SL[1G] uses LTL syntax augmented with an extra rule $\wp\flat\varphi$, where $\wp$ is a quantification prefix (e.g. $[\![e]\!]\langle\!\langle x\rangle\!\rangle[\![y]\!]$), $\flat$ is a binding prefix (e.g. $(\mathrm{E}, e)(a, x)(b, y)$), and φ is an LTL formula. We first define the concept of a quantification and binding prefix formally [72].

[1]However, this is not the case for certain fragments of ATL* like LTL, which is PSPACE-COMPLETE with respect to the size of the formula (see Table 2.1). While our algorithm can be used for model checking LTL, it is far from optimal.

Definition 5.1 (Sl[1g] Prefixes)**.** A *quantification prefix* over a set of variables $V \subseteq \mathit{Var}$ is a finite word $\wp \in \{\langle\!\langle x \rangle\!\rangle, [\![x]\!] \mid x \in V\}^{|V|}$ of length $|V|$ such that each variable $x \in V$ occurs in $\wp$ exactly once. QPre_V denotes the set of all quantification prefixes over V. $\mathit{QPre} \triangleq \bigcup_{V \subseteq \mathit{Var}} \mathit{QPre}_V$ is the set of all quantification prefixes.

A *binding prefix* over a set of variables $V \subseteq \mathit{Var}$ is a finite word $\flat \in \{(a, x) \mid a \in \mathit{Agt} \wedge x \in V\}^{|\mathit{Agt}|}$ of length $|\mathit{Agt}|$ such that each agent $i \in \mathit{Agt}$ occurs in $\flat$ exactly once. BPre_V denotes the set of all binding prefixes over V. $\mathit{BPre} \triangleq \bigcup_{V \subseteq \mathit{Var}} \mathit{BPre}_V$ is the set of all binding prefixes.

For example, $\wp_1 = \langle\!\langle z \rangle\!\rangle [\![x]\!] \langle\!\langle y \rangle\!\rangle$ is a valid quantification prefix over the set of variables $V = \{x, y, z\}$, i.e. $\wp_1 \in \mathit{QPre}_V$. However, $\wp_2 = \langle\!\langle x \rangle\!\rangle \langle\!\langle y \rangle\!\rangle$ and $\wp_3 = \langle\!\langle x \rangle\!\rangle [\![y]\!] [\![z]\!] \langle\!\langle x \rangle\!\rangle$ are not valid quantification prefixes because $\wp_2$ does not quantify $z \in V$ and $\wp_3$ contains $x \in V$ twice (also their lengths are not $|V| = 3$). Similarly, $\flat_1 = (b, x)(a, x)$ is a valid binding prefix over V when the set of agents is $\mathit{Agt} = \{a, b\}$, whereas $(a, x)(a, y)$ and $(a, x)(b, u)$ are not. Note that $\mathit{BPre} = \mathit{BPre}_{\mathit{Var}}$ by construction.

We are now ready to provide a formal definition of Sl[1g] syntax [72].

Definition 5.2 (Sl[1g] Syntax)**.** Sl[1g] *formulas* are built inductively from the set of atomic propositions AP, strategy variables Var, agents Agt, quantification prefixes QPre, and binding prefixes BPre, by using the following grammar, with $p \in AP$, $x \in \mathit{Var}$, $a \in \mathit{Agt}$, $\flat \in \mathit{BPre}$, and $\wp \in \mathit{QPre}$:

$$\varphi ::= p \mid \top \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \mathsf{F}\,\varphi \mid \mathsf{G}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi \mid \wp\flat\varphi$$

where $\wp \in \mathit{QPre}_{\mathsf{free}(\flat\varphi)}$. *SL[1G]* denotes the infinite set of formulas generated by the above rules.

The conditions on $\wp$ and $\flat$ above ensure that $\wp\flat\varphi$ is an Sl[1g] sentence, i.e. $\mathsf{free}(\wp\flat\varphi) = \emptyset$. We must simply make sure that each agent is bound to a variable in $\flat$ and that the variable is quantified in $\wp$. Assume that the set of agents is $\mathit{Agt} = \{a, b\}$. Then $[\![x]\!] \langle\!\langle y \rangle\!\rangle (a, x)(b, y) \mathsf{F}\,\mathsf{G}\,p$ is a well-formed Sl[1g] formula (in fact, it is an Sl[1g] sentence). On the other hand, $[\![x]\!](a, x)q$ is not well-formed because it does not bind agent b to any strategy variable. A prefix is sometimes more readable when it interleaves quantifiers and bindings (e.g. $[\![x]\!](a, x)\langle\!\langle y \rangle\!\rangle(b, y)\mathsf{X}\,r$). While this violates the definition of Sl[1g] syntax, we will allow it as long as each binding occurs after the corresponding quantifier (e.g. $(a, x)\langle\!\langle x \rangle\!\rangle(b, x)\,s$ is not a well-formed Sl[1g] formula because (a, x) appears before $\langle\!\langle x \rangle\!\rangle$).

Sl[1g] is defined with respect to memoryful strategies. As we have explained in Chapter 3, incomplete information is undecidable under perfect recall semantics. Therefore, we have (again) two options: *(i)* consider Sl[1g] with imperfect recall, or *(ii)* consider Sl[1g] with complete information. The first option turns out not to be very interesting because we obtain a logic which is strictly less expressive than Slk while most probably still having the same complexity[2], namely PSpace with respect to both the size of the model and the formula. Therefore, we consider Sl[1g] with *perfect recall* and *complete information.* Consequently, Sl[1g] uses *shared (memoryful) strategies* (Definition 2.21) instead of uniform shared memoryless strategies (Definition 4.2). Finally, since Sl[1g] is a syntactic fragment of Sl, the semantics remains the same (Definition 2.29).

5.2 Model Checking

In this section, we present a novel model checking algorithm for Sl[1g], which we obtain by reducing the model checking problem to solving an infinite two-player parity game. The structure of this section is as follows: Subsection 5.2.1 introduces the algorithm based on the reduction, Subsection 5.2.2 discusses the complexity of the algorithm, Subsection 5.2.3 explains how the algorithm can be modified to support general strategy synthesis, Subsection 5.2.4 describes an efficient symbolic implementation of the algorithm, and Subsection 5.2.5 presents an optimisation technique for the algorithm called separate determinisation.

[2]There is a chance that a more efficient algorithm might be found for Sl[1g] with imperfect recall. However, recall that although Atl with imperfect recall is not completely subsumed by Sl[1g] with imperfect recall (see Subsection 4.2.1), it is less expressive. Since the complexity of model checking Atl with imperfect recall is P^{NP}-complete [23], there is most likely very little to be gained.

5.2.1 Algorithm

We introduce here the new algorithm for model checking arbitrary SL[1G] formulas. Consider an interpreted system $\mathcal{I}$ and an SL[1G] sentence $\varphi \in$ *SL[1G]*. To make the algorithm easier to understand, we assume that φ is a *principal sentence* of the form $\wp\flat\psi$ with $\wp \in QPre_{\mathsf{free}(\flat\psi)}$ and $\flat \in BPre$. If this is not the case, we can simply add one quantifier and agent binding for each agent (it will not matter because φ is a sentence). Since our discussion will be based largely on SL[1G] principal sentences, we shall define the concept formally.

Definition 5.3 (SL[1G] Principal Sentence)**.** Let $\mathcal{I}$ be an interpreted system. An SL[1G] formula $\varphi \in$ *SL[1G]* is a *principal sentence* iff *(i)* it is a sentence and *(ii)* it is of the form $\wp\flat\psi$ where $\wp \in QPre_{\mathsf{free}(\flat\psi)}$ is a quantification prefix, $\flat \in BPre$ a binding prefix, and $\psi \in$ *SL[1G]* an SL[1G] formula. Furthermore, φ is a *basic principal sentence* iff $\psi \in LTL$ is an LTL formula.

An SL[1G] formula $\varphi' \in$ *SL[1G]* is a *principal subsentence* of φ iff φ' is a principal sentence and a subformula of φ. Furthermore, φ' is a *strict principal subsentence* of φ iff φ' is a principal subsentence of φ and $\varphi' \neq \varphi$. Finally, φ' is a *direct principal subsentence* of φ iff *(i)* φ' is a strict principal subsentence of φ and *(ii)* there exists no strict principal subsentence φ'' of φ such that φ' is a strict principal subsentence of φ''.

Although we introduced a considerable number of new terms in the definition above, they should be easy to understand. To make sure that the precise meanings of all the concepts are clear, we provide a single comprehensive example[3]:

$$\varphi = [\![x]\!](1,x)(2,x)\mathsf{G}\,[\neg p_1 \wedge \neg p_2 \wedge \underbrace{[\![y]\!]\langle\!\langle z\rangle\!\rangle(1,y)(2,z)\mathsf{X}\,(p_2 \wedge \neg\,\overbrace{\langle\!\langle u\rangle\!\rangle\langle\!\langle v\rangle\!\rangle(1,u)(2,v)\mathsf{F}\,\neg p_2}^{\varphi''})}_{\varphi'}]$$

There are three principal sentences, namely φ, φ', and φ''. φ'' is also the only *basic* principal sentence (both φ and φ' have non-LTL subformulas). The principal subsentences of φ are φ, φ', and φ''. However, only φ' and φ'' are *strict* principal subsentences of φ. Finally, both φ and φ' each have exactly one *direct* principal subsentence, namely φ' and φ'' (although in general a formula might have several direct principal subsentences). Note that φ'' cannot have any strict or direct principal subsentences because it is a basic principal sentence.

Recursive Approach

Our aim is now to find the set of all states $\|\varphi\|_{\mathcal{I}} \subseteq G$ at which an SL[1G] principal sentence $\varphi = \wp\flat\psi$ holds, i.e. $\|\varphi\|_{\mathcal{I}} \triangleq \{g \in G \mid \mathcal{I},\emptyset,g \models_{\mathrm{SL}} \varphi\}$. We proceed in a recursive manner over the structure of φ. According to SL[1G] syntax (Definition 5.2), ψ is a formula which combines atoms and *direct principal subsentences* of the form $\varphi' = \wp'\flat'\psi'$ using only Boolean and temporal connectives. For example:

$$\varphi = \underbrace{\langle\!\langle x\rangle\!\rangle[\![y]\!]}_{\wp}\underbrace{(a,x)(b,y)}_{\flat}\underbrace{[p \to \mathsf{X}\ \overbrace{[\![u]\!][\![v]\!]}^{\wp'}\overbrace{(a,u)(b,v)}^{\flat'}\overbrace{\mathsf{F}\,\mathsf{G}\,q}^{\psi'}]}_{\psi}$$

Since φ' is a principal subsentence, we have $\mathsf{free}(\varphi') = \emptyset$. Therefore, for all $g \in G$ and $\chi \in Asg$, we have $\mathcal{I},\chi,g \models_{\mathrm{SL}} \varphi'$ iff $\mathcal{I},\emptyset,g \models_{\mathrm{SL}} \varphi'$. Using our definition of $\|\varphi'\|_{\mathcal{I}}$, we get $\mathcal{I},\chi,g \models_{\mathrm{SL}} \varphi'$ iff $g \in \|\varphi'\|_{\mathcal{I}}$ for all $g \in G$ and $\chi \in Asg$. In other words, φ' holds in all states $\|\varphi'\|_{\mathcal{I}}$ (and no other states) regardless of the assignment χ. Therefore, we can do the following for each direct principal subsentence $\varphi' = \wp'\flat'\psi'$ of φ:

1. Calculate $\|\varphi'\|_{\mathcal{I}}$ (recursively);
2. Replace φ' in φ with a new atom $p_{\varphi'} \in AP$;
3. Update the assignment $h := h \cup \{(p_{\varphi'}, \|\varphi'\|_{\mathcal{I}})\}$.

[3]The example is based on the toy model in Figure 3.1. We omit strategies for the environment for conciseness (it does not matter whether we use existential or universal quantifiers anyway). The meaning of the formula is roughly: *"As long as both players perform the same actions, neither of them wins and for each player 1 strategy there is a player 2 strategy which ensures that player 2 wins in the next round and nothing can take his victory away after that"*.

This procedure preserves the truth value of φ. The following lemma asserts this formally.

Lemma 5.1. Let $\mathcal{I}$ be an interpreted system and $\varphi = \wp\flat\psi$ an arbitrary SL[1G] principal sentence. Furthermore, let $\varphi^* = \wp\flat\psi^*$ be the principal sentence with all direct principal subsentences φ' of φ replaced with atoms $p_{\varphi'}$ such that $h^*(p_{\varphi'}) \triangleq \|\varphi'\|_{\mathcal{I}}$ and $\mathcal{I}^*$ the interpreted system with the updated assignment h^*. Then for all global states $g \in G$ and assignments $\chi \in Asg$, we have $\mathcal{I}, \chi, g \models_{\text{SL}} \varphi$ iff $\mathcal{I}^*, \chi, g \models_{\text{SL}} \varphi^*$.

Proof (Sketch). The proof is performed by induction on the structure of φ. We show here only the case when two corresponding subformulas of φ and φ^* are different, i.e. when a principal subsentence φ' is replaced with a new atom $p_{\varphi'}$. Let φ' be an arbitrary principal subsentence of ψ. Then we have for all states $g \in G$ and assignments $\chi \in Asg$:

$$\begin{aligned}
\mathcal{I}, \chi, g \models_{\text{SL}} \varphi' &\text{ iff } \mathcal{I}, \emptyset, g \models_{\text{SL}} \varphi' \\
&\text{ iff } g \in \|\varphi'\|_{\mathcal{I}} \\
&\text{ iff } g \in h^*(p_{\varphi'}) \\
&\text{ iff } \mathcal{I}^*, \emptyset, g \models_{\text{SL}} p_{\varphi'} \\
&\text{ iff } \mathcal{I}^*, \chi, g \models_{\text{SL}} p_{\varphi'}
\end{aligned}$$

The first and the last equivalence follow from the fact that φ' and $p_{\varphi'}$ are SL[1G] sentences. The second one is the very definition of $\|\varphi'\|_{\mathcal{I}}$. The third one is true by construction of h^*. The fourth one follows from SL semantics (Definition 2.29). □

Once we have replaced all principal subsentences of ψ, we are left with an SL[1G] *basic principal sentence* $\varphi^* = \wp\flat\psi^*$ where $\psi^* \in LTL$ is an LTL formula. Hence, we have reduced the problem of model checking an arbitrary SL[1G] *principal sentence* to recursive model checking of SL[1G] *basic principal sentences.* We shall now see how to solve this simpler problem.

Algorithm Overview

We outline here how the problem of model checking SL[1G] basic principal sentences $\varphi = \wp\flat\psi$ can be reduced to the problem of solving a two-player parity game. Let $\mathcal{I}$ be an interpreted system. To calculate the set of global states at which ψ holds in $\mathcal{I}$, we proceed as follows:

1. **Formula automaton.** We construct a deterministic parity automaton $\mathfrak{P}^{\psi}_{\mathcal{I}}$ equivalent to the LTL formula ψ.
2. **Arena construction.** We construct a two-player arena $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ representing the global state space G and the interdependency of strategies in the prefix $\wp\flat$.
3. **Game combination.** We combine the arena $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ and parity automaton $\mathfrak{P}^{\psi}_{\mathcal{I}}$ into a two-player parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$. Solving the parity game yields its winning regions, which can be used to calculate $\|\varphi\|_{\mathcal{I}}$.

We shall now expand on each of the steps above. Unlike the SLK model checking algorithm we proposed in Subsection 4.2.2, our new SL[1G] model checking algorithm relies on the theory of ω-automata and infinite games. Please refer to Section 2.4 for a brief introduction into these areas.

Formula Automaton

The first step of our new model checking algorithm for SL[1G] is the construction of a deterministic parity automaton equivalent to the underlying LTL formula ψ of an SL[1G] basic principal formula $\varphi = \wp\flat\psi$. This is a very standard procedure, which is usually performed in three steps:

1. We construct a *non-deterministic generalised Büchi automaton* $\mathfrak{A}_{\psi}$ equivalent to ψ. We do this using the standard translation presented in Figure 2.6. An explanation of the procedure with examples is provided in Subsection 2.4.3.

2. We convert $\mathfrak{A}_\psi$ to a *non-deterministic Büchi automaton* $\mathfrak{B}_\psi$. As explained in Subsection 2.4.1, this can be done easily using an automaton product with the deterministic automaton shown in Figure 2.5.

3. We transform $\mathfrak{B}_\psi$ into an equivalent *deterministic parity automaton* $\mathfrak{P}_\psi$ using the determinisation procedure outlined in Subsection 2.4.4.

The automata $\mathfrak{A}_\psi$, $\mathfrak{B}_\psi$, and $\mathfrak{P}_\psi$ above use sets of atomic propositions $A \in 2^{AP}$ as their alphabet. Given an interpreted system $\mathcal{I}$, we can convert them into automata $\mathfrak{A}^\psi_\mathcal{I}$, $\mathfrak{B}^\psi_\mathcal{I}$, and $\mathfrak{P}^\psi_\mathcal{I}$ on global states G by modifying their transition relations accordingly:

$$R_\mathcal{I}(s,g,s') \quad \text{iff} \quad R(s,\{p \in AP \mid g \in h(p)\},s') \quad \text{for all} \quad (s,g,s') \in S \times G \times S'$$

It is easy to see that $\mathfrak{A}^\psi_\mathcal{I}$, $\mathfrak{B}^\psi_\mathcal{I}$, and $\mathfrak{P}^\psi_\mathcal{I}$ accept all infinite paths $\pi \in \mathit{Pth}$ in $\mathcal{I}$ which satisfy ψ, i.e. $\mathcal{I},\pi \models_{\text{LTL}} \psi$. Note that the resulting automata depend only on the model $\mathcal{I}$ and the underlying Ltl formula ψ, i.e. they are independent of the prefixes $\wp$ and $\flat$. Nevertheless, we will sometimes refer to them as $\mathfrak{A}^\varphi_\mathcal{I}$, $\mathfrak{B}^\varphi_\mathcal{I}$, and $\mathfrak{P}^\varphi_\mathcal{I}$ because it simplifies the notation.

To give the reader a more concrete idea of Sl[1g] model checking and strategy synthesis, we will now introduce an example, which we will use several times to demonstrate key concepts. Consider the toy model in Figure 3.1 and the Sl[1g] basic principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E},e)(1,x)(2,y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$, which means roughly: *"Whichever action player 1 performs, there exists an action for player 2 such that neither player will ever win"*. The quantification and binding prefixes are $\wp = [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle$ and $\flat = (\mathrm{E},e)(1,x)(2,y)$ respectively. The underlying Ltl formula is $\psi = \mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. The deterministic parity automaton $\mathfrak{P}^\gamma_\mathcal{I}$ equivalent to ψ in $\mathcal{I}$ is shown in Figure 5.1a. We obtained it using the procedure outlined above. Note that it is not minimal because the states t_I and t_1 could be merged (with colour 0).

We would like to point out that the recursive step, i.e. replacing direct principal subsentences φ' of φ with atoms $p_{\varphi'}$, can be incorporated into the procedure for constructing the non-deterministic generalised Büchi automaton $\mathfrak{A}^\psi_\mathcal{I}$. Assume that we are currently model checking an Sl[1g] principal sentence $\varphi = \wp\flat\psi$ where ψ is not necessarily an Ltl formula. We can construct the non-deterministic generalised Büchi automaton $\mathfrak{A}^\varphi_\mathcal{I}$ straightaway (despite ψ possibly not being an Ltl formula) by adding an extra case to the standard translation in Figure 2.6:

```
function GenBüchi($\Phi$)
    switch $\Phi$ do
        ...
        case $\wp'\flat'\psi'$:
            $h := h \cup \{(p_\Phi, \|\Phi\|_\mathcal{I})\}$        ▷ Solve recursively and update assignment.
            $\Phi := p_\Phi$                                 ▷ Replace subsentence with the new atom.
            return $\mathcal{A}_\exists(\emptyset, AP, p_\Phi, \top, \emptyset)$
        ...
    end switch
end function
```

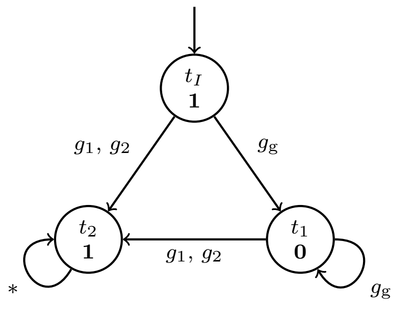
Hence, we actually do not need to transform φ to an Sl[1g] basic principal sentence separately because the modified procedure will take care of all direct principal subformulas $\wp'\flat'\psi'$ automatically.

Arena Construction

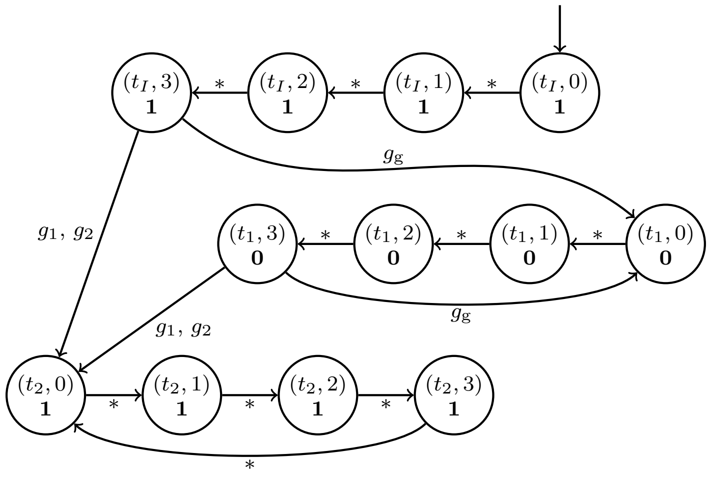
The main idea of the whole transformation is to convert the model checking problem to a two-player game (see Subsection 2.4.5 for a brief introduction to games). Intuitively, model checking an Sl[1g] basic principal sentence $\varphi = \wp\flat\psi$, i.e. deciding whether $\mathcal{I},\emptyset,g \models_{\text{SL}} \varphi$ for some global state $g \in G$, can be imagined as a simple finite game between two players. The players take turns depending on the order of quantifiers in $\wp$ and select the (whole) strategies for each variable in the quantifier:

- The *existential player*, selecting strategies for existentially quantified variables (e.g. $\langle\!\langle x\rangle\!\rangle$), is trying to satisfy the formula ψ;

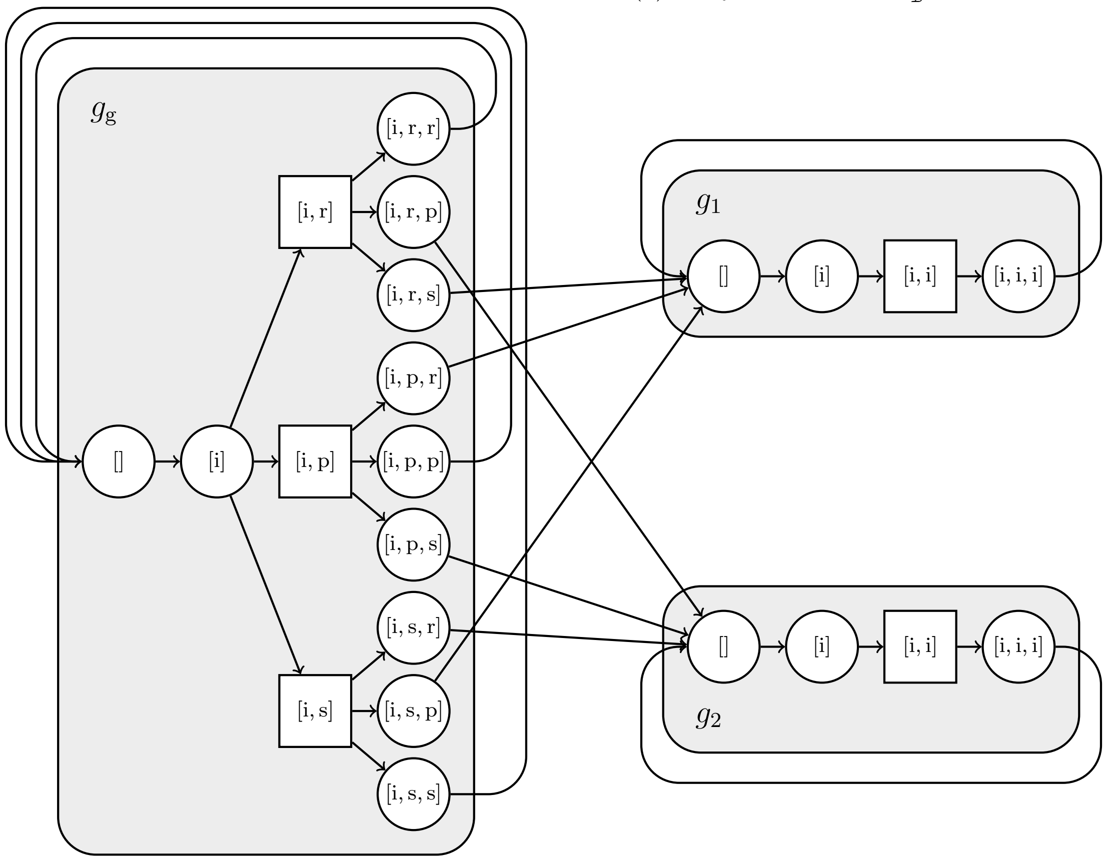
- The *universal player*, selecting strategies for universally quantified variables (e.g. $[\![y]\!]$), is trying to falsify the formula ψ.



(a) Parity automaton $\mathfrak{P}^{\gamma}_{\mathcal{I}}$.



(b) Delayed automaton $\mathfrak{D}^{\gamma}_{\mathcal{I}}$.



(c) Formula arena $\mathcal{A}^{\gamma}_{\mathcal{I}}$. The existential and universal player states are represented by squares and circles respectively.

Figure 5.1: Parity automaton $\mathfrak{P}^{\gamma}_{\mathcal{I}}$, delayed automaton $\mathfrak{D}^{\gamma}_{\mathcal{I}}$, and formula arena $\mathcal{A}^{\gamma}_{\mathcal{I}}$ of the toy model for the Sl[1G] basic principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\ [\neg p_1 \wedge \neg p_2]$. A star represents an arbitrary global state.

For example, if $\wp = \langle\langle x\rangle\rangle[\![y]\!]\langle\langle z\rangle\rangle$, then the game is as follows: the existential player selects a strategy for x, the universal player selects a strategy for y, and finally the existential player selects a strategy for z. Once all strategies have been selected, we construct the corresponding complete assignment $\chi \in \mathit{CAsg}$ (from the players' choices and the binding prefix $\flat$), and check the Ltl formula ψ. Given that the way in which the strategies were selected by the players is in accordance with Sl semantics (Definition 2.29), we have $\mathcal{I}, \emptyset, g \models_{\mathrm{SL}} \varphi$ iff $\mathcal{I}, \chi, g \models_{\mathrm{SL}} \psi$.

The finite game we outlined above is very simple and short. In fact, it will have exactly $|\wp|$ rounds, one for each strategy variable. However, there are possibly infinitely many strategies and hence infinitely many moves that one of the players can make in each round. We see that the finite game idea does not make our problem any easier. Ideally, we would want the players to make simple choices (e.g. select a single action) as the game progresses. Instead of selecting the whole strategies up front, they would somehow build them gradually throughout the game. After all, this is much closer to the concept of a strategy in "games" that people play for fun. The question is now: Is such a reduction possible for Sl[1g] model checking?

The answer is "yes", we can reduce Sl[1g] model checking of φ to solving an infinite two-player game where the players pick an action for each strategy in a given state. We will explain this reduction in more detail shortly. We first have to justify why we can do this. It turns out that this is the very reason why *behavioural semantics* of Sl[1g] is so important. It was shown in [75] that for an arbitrary Sl (including Sl[1g]) sentence $\wp\psi_{\mathrm{SL}} \in \mathit{SL}$ where $\wp \in \mathit{QPre}_{\mathsf{free}(\psi_{\mathrm{SL}})}$ and ψ_{SL} is an agent-closed Sl formula, we have for each global state $g \in G$, $\mathcal{I}, \emptyset, g \models_{\mathrm{SL}} \wp\psi_{\mathrm{SL}}$ iff there exists a dependence map θ_{SL} for $\wp$ over strategies such that $\mathcal{I}, g, \theta_{\mathrm{SL}}(\chi) \models_{\mathrm{SL}} \psi_{\mathrm{SL}}$ for all $\chi \in \mathit{Asg}$ with $[\![\wp]\!] = \mathrm{dom}(\chi)$. A dependence map θ for the quantification prefix $\wp$ over strategies is a function from strategies for universally quantified variables in $\wp$ (denoted $[\![\wp]\!]$) to strategies for all quantified variables in $\wp$ (denoted $\wp$):

$$\Bigg([\![\wp]\!] \to \underbrace{\Big(\mathit{Trk} \to \bigcup_{i\in \mathit{Agt}} \mathit{Act}_i\Big)\Bigg)}_{\substack{\text{strategies for } \textit{universally}\\ \textit{quantified} \text{ variables}}} \to \Bigg(\wp \to \underbrace{\Big(\mathit{Trk} \to \bigcup_{i\in \mathit{Agt}} \mathit{Act}_i\Big)\Bigg)}_{\substack{\text{strategies for}\\ \textit{all} \text{ variables}}}$$

which satisfies the independence of variables. For example, if $\wp = [\![x]\!]\langle\langle y\rangle\rangle[\![z]\!]$ then y is independent of z (because it appears before it) and x, z are independent of all variables (because they are universally quantified). This must be ensured by the dependence map (e.g. $\theta_{\mathrm{SL}}(\chi)(z) = \chi(z)$). The problem with θ_{SL} is that the action of a strategy in one game might depend on the action of another strategy in another game (non-behavioural semantics). Fortunately, a stronger result holds for Sl[1g] principal sentences $\wp\flat\psi_{\mathrm{SL[1G]}}$ [74]: There exists an *elementary* dependence map $\theta_{\mathrm{SL[1G]}}$ for $\wp$ over strategies (iff the principal sentence holds in the state). A dependence map is elementary iff it can be transformed into an equivalent function (called *adjoint*) of the form:

$$\mathit{Trk} \to \Bigg(\Big([\![\wp]\!] \to \underbrace{\bigcup_{i\in \mathit{Agt}} \mathit{Act}_i\Big)}_{\substack{\text{next actions}\\ \text{for } \textit{universally}\\ \textit{quantified} \text{ variables}}} \to \Big(\wp \to \underbrace{\bigcup_{i\in \mathit{Agt}} \mathit{Act}_i\Big)}_{\substack{\text{next actions}\\ \text{for } \textit{all} \text{ variables}}}\Bigg)$$

Intuitively, this means that given the history of a play *Trk*, the next actions of strategies depend only on the next actions of the universally quantified strategies. This is exactly what we want! Before we return to the reduction, it should be pointed out that this (existence of an elementary dependence map) is the very definition of behavioural semantics[4]. Our explanation of dependence maps was quite informal. Please refer to [74, 75] for a more formal explanation and proofs.

We can therefore reduce the model checking of an Sl[1g] basic principal sentence $\wp\flat\psi$ to solving the following infinite game: Let $g \in G$ be the current global state. For each $0 \leq k < |\wp|$ (in increasing order), the existential or universal player selects an action $a_k \in \mathit{Act}_{\mathsf{sharing}(\flat\psi,\wp(k))}$ for the strategy assigned to

[4]Formally, behavioural semantics are defined as follows [75]: Let $\mathcal{I}$ be an interpreted system and $\varphi = \flat\psi$ an Sl sentence with $\wp \in \mathit{QPre}_{\mathsf{free}(\psi)}$. Then for each global state $g \in G$, we have $\mathcal{I}, \emptyset, g \models_{\mathrm{SLB}} \varphi$ iff there exists an elementary dependence map θ for the quantification prefix $\wp$ over strategies such that $\mathcal{I}, g, \theta(\chi) \models_{\mathrm{SL}} \psi$ for all $\chi \in \mathit{Asg}$ with $[\![\wp]\!] = \mathrm{dom}(\chi)$. Notice the two different modelling relations "$\models_{\mathrm{SLB}}$" (behavioural semantics) and "$\models_{\mathrm{SL}}$" (Sl semantics).

$\wp(k)$. Once all actions have been selected, a temporal transition according to the resulting joint action $a \in Act$ is performed and the new current state is $g' = t(g, a)$. This pattern of selecting actions and performing a temporal transition is repeated forever. Let $\pi = gg' \ldots \in Pth$ be the infinite path obtained by this procedure. We define that the existential player wins the game iff the LTL formula ψ holds along π, i.e. $\mathcal{I}, \pi \models_{\text{LTL}} \psi$. The behavioural semantics of SL[1G] then imply:

$$\mathcal{I}, \emptyset, g \models_{\text{SL}} \wp\flat\psi \quad \text{iff} \quad g \text{ is in the winning region of the existential player}$$

Essentially, we have just reduced SL[1G] model checking to solving a two-player infinite game with an LTL winning condition.

There is one rather cosmetic issue with our formulation of the game, namely the fact that multiple decisions are made by both players within the same state (selecting actions for all strategies) before a move occurs. We shall make it more standard by inflating the state space and constructing an *arena* whose states are pairs (g, d) where $g \in G$ is the current state and d is a tuple of actions selected so far. Initially, d is empty and is gradually appended by the players as they choose actions for strategies. Once all actions have been selected, d represents a joint action, a temporal transition occurs, and the new state is $(t(g, d), [])$. A formal definition of the formula arena follows.

Definition 5.4 (Formula Arena)**.** Let $\mathcal{I}$ be an interpreted system and $\varphi \in$ *SL[1G]* a SL[1G] principal sentence of the form $\wp\flat\psi$. We construct a *formula arena* $\mathcal{A}_{\mathcal{I}}^{\wp\flat} = (V_0, V_1, E)$ of $\mathcal{I}$ for φ where:

- The states V of $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ are pairs $(g, d) \in G \times Dec_{\mathcal{I}}^{\wp\flat}$ such that for all $0 \le i < |d|$ we have $d(i) \in \bigcap_{i \in \mathsf{sharing}(\flat\top, \wp(k))} P_i(l_{i\mathrm{E}}(g))$
- The existential player states $V_0 \subseteq V$ are states $(g, d) \in V$ such that $|d| < |\wp|$ and $\wp(|d|)$ is an existential strategy quantifier.
- The universal player states $V_1 \subseteq V$ are states $(g, d) \in V$ such that $|d| = |\wp|$ or $\wp(|d|)$ is a universal strategy quantifier.
- The edge relation is defined as:

$$E = \{((g, d), (g, d \cdot a)) \in V \times V \mid |d| < |\wp|\} \cup \left\{\left((g, d), \left(t(g, d^{Act}), []\right)\right) \in V \times V \;\middle|\; |d| = |\wp|\right\}$$

where $Dec_{\mathcal{I}}^{\wp\flat} \triangleq \bigcup_{\ell=0}^{|\wp|} \prod_{k=0}^{\ell-1} \bigcup_{i \in \mathsf{sharing}(\flat\top, \wp(k))} Act_i$ is the set of decisions and $d^{Act} \in Act$ is a joint action such that for all $0 \le k < |\wp|$ and $i \in \mathsf{sharing}(\flat\top, \wp(k))$ we have $a_i(d^{Act}) = d(k)$.

Note that we could allow decisions of length at most $|\wp| - 1$ (instead of $|\wp|$) and perform temporal transition immediately when the action for the last strategy is selected. The reason why we chose to include the extra step is that it nicely models non-determinism in case we use an evolution *relation* (instead of an evolution function). Intuitively, this is the same as adding an extra universally quantified variable at the end of $\wp$ representing the final evolution choice. Again, we shall sometimes refer to a formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ as $\mathcal{A}_{\mathcal{I}}^{\wp\flat\psi}$ although it is independent of the underlying LTL formula ψ. Finally, notice that the definition uses the identity $\mathsf{sharing}(\flat\top, \wp(k)) = \mathsf{sharing}(\flat\psi, \wp(k))$ which can be shown for all SL[1G] principal sentences $\wp\flat\psi$ and indices $0 \le k < |\wp|$.

For example, the formula arena $\mathcal{A}_{\mathcal{I}}^{\gamma}$ of the toy model in Figure 3.1 and the SL[1G] principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$ introduced earlier is shown in Figure 5.1c. Let $f \in SStr_{\{\mathrm{E}\}}$, $g \in SStr_{\{1\}}$, and $h \in SStr_{\{2\}}$ be the underlying strategies of the agents. Consider a game that starts in the global state g_{g}. The corresponding state of the arena is $(g_{\mathrm{g}}, [])$, which is a universal player state corresponding to the quantifier $[\![e]\!]$. Only one action is available to the environment so the universal player must move to the state $(g_{\mathrm{g}}, [\mathrm{i}])$, i.e. set $f(g_{\mathrm{g}}) = \mathrm{i}$. Now it is again the universal player's turn. He has three options for $[\![x]\!]$, namely r, p, and s. Suppose he picks r. Thus, we move to the state $(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{r}])$ and have $g(g_{\mathrm{g}}) = \mathrm{r}$. The existential player now has the same three options for $\langle\!\langle y\rangle\!\rangle$. Suppose he picks s. We end up in the state $(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{r}, \mathrm{s}])$ and have $h(g_{\mathrm{g}}) = \mathrm{s}$. Since all actions have been selected, we convert the decision $[\mathrm{i}, \mathrm{r}, \mathrm{s}]$ to the joint action $a = \mathrm{rs}$. As we are using an evolution function, there is exactly one next state that the universal player must move to, namely $(t(g_{\mathrm{g}}, a), []) = (g_1, [])$. The whole process starts again. The universal player has to choose the action i for $[\![e]\!]$ so we move to the state $(g_1, [\mathrm{i}])$ and set $f(g_{\mathrm{g}} g_1) = \mathrm{i}$. Then the universal player has to choose an action for $[\![x]\!]$ and so on. We hope that this detailed example gives the reader a good understanding of the formula arena concept.

It remains to explain what the *winning condition* of the new game is and how it relates to Sl[1G] model checking. Intuitively, the existential player wins an infinite play $\left(g_0, d_0^0\right) \dots \left(g_0, d_0^{|\wp|}\right) \left(g_1, d_1^0\right) \dots \in V^\omega$ in the formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ for an Sl[1G] basic principal sentence $\wp\flat\psi$ iff the infinite path $g_0 g_1 \dots \in \mathit{Pth}$ satisfies the Ltl formula ψ. The following definition expresses this formally.

Definition 5.5 (Pseudo-Ltl Game)**.** Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi$ an Sl[1G] basic principal sentence. The *pseudo*-Ltl *game* $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ of $\mathcal{I}$ for $\wp\flat\psi$ is a game $\left(\mathcal{A}_{\mathcal{I}}^{\wp\flat}, \mathit{Win}\right)$ where $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ is the formula arena of $\mathcal{I}$ for $\wp\flat\psi$ and $\mathit{Win} \subseteq V^\omega$ is the winning set which contains all possible paths $\pi \in V^\omega$ such that $\mathcal{I}, \pi_{\mathcal{I}} \models_{\mathrm{LTL}} \psi$. $\pi_{\mathcal{I}} \in \mathit{Pth}$ refers to the underlying path in $\mathcal{I}$, i.e. $\forall i \geq 0 \exists d \in \mathit{Dec}_{\mathcal{I}}^{\wp\flat}. (\pi_{\mathcal{I}}(i), d) = \pi((|\wp| + 1)i)$.

Informally, $\pi_{\mathcal{I}}$ is equal to π "modulo" $(|\wp| + 1)$ without the decisions. Consider again the formula arena $\mathcal{A}_{\mathcal{I}}^{\gamma}$ in Figure 5.1c of the toy model in Figure 3.1 and the Sl[1G] principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. Let $\pi \in V^\omega$ be an infinite path in $\mathcal{A}_{\mathcal{I}}^{\gamma}$ which stays forever within the box labelled g_{g}, e.g. $\pi = (g_{\mathrm{g}}, [])\,(g_{\mathrm{g}}, [\mathrm{i}])\,(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{p}])\,(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{p}, \mathrm{p}])\,(g_{\mathrm{g}}, [])\,(g_{\mathrm{g}}, [\mathrm{i}])\,(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{s}])\,(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{s}, \mathrm{s}]) \dots$. The underlying path in $\mathcal{I}$ is $\pi_{\mathcal{I}} = g_{\mathrm{g}} g_{\mathrm{g}} \dots = g_{\mathrm{g}}^\omega$. Since $\mathcal{I}, \pi_{\mathcal{I}} \models_{\mathrm{LTL}} \mathsf{G}\,[\neg p_1 \wedge \neg p_2]$, π is winning in $\mathfrak{L}_{\mathcal{I}}^{\gamma}$ by the definition above.

The important relationship between pseudo-Ltl games and Sl[1G] model checking is stated in the following lemma.

Lemma 5.2. Let $\mathcal{I}$ be an interpreted system and $\varphi = \wp\flat\psi$ an Sl[1G] basic principal sentence. Then for all global states $g \in G$ we have: $\mathcal{I}, \emptyset, g \models_{\mathrm{SL}} \varphi$ iff $(g, [])$ is a winning state for the existential player in the pseudo-Ltl game $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$.

Proof (Sketch). The problem of model checking Sl[1G] can be reduced to solving a so-called *dependence-vs-valuation game* [72]. In this game, the players alternate as follows: the existential player chooses a dependence map $\theta : \left([\![\wp]\!] \to \bigcup_{i \in \mathit{Agt}} \mathit{Act}_i\right) \to \left(\wp \to \bigcup_{i \in \mathit{Agt}} \mathit{Act}_i\right)$ for $\wp$ over actions in the current state $g \in G$. Then the universal player chooses a valuation $v : [\![\wp]\!] \to \bigcup_{i \in \mathit{Agt}} \mathit{Act}_i$. The combination $\theta(v) : \wp \to \bigcup_{i \in \mathit{Agt}} \mathit{Act}_i$ assigns actions to all variables and determines the next state $g' \in G$. The existential player wins the game iff the infinite path $\pi = gg' \dots \in \mathit{Pth}$ satisfies the Ltl formula ψ.

Instead of choosing the whole dependence map and valuation at once, the players in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ assign actions to strategies one by one for each quantifier in $\wp$. The order of players' moves in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ ensures that the independence constraints of the dependence map are satisfied (while not imposing any extra restrictions). Given this and the fact that the winning conditions are the same, the two games are equivalent. Hence, our claim follows. □

Game Combination

We now have all the ingredients to construct the parity game. Let $\mathcal{I}$ be an interpreted system and $\varphi = \wp\flat\psi$ an Sl[1G] *basic* principal sentence. We have done the following so far:

1. We constructed a deterministic parity automaton $\mathfrak{P}_{\mathcal{I}}^{\psi} = (T, G, t_I, \delta, c)$ which accepts exactly those paths which satisfy the Ltl formula ψ.

2. We constructed a formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat} = (V_0, V_1, E)$ of $\mathcal{I}$ for φ. The states of the arena are possible pairs of global states and decisions $V \subseteq G \times \mathit{Dec}_{\mathcal{I}}^{\wp\flat}$. The winning set of the corresponding pseudo-Ltl game $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ contains all infinite paths $\pi \in V^\omega$ such that the underlying path in the interpreted system $\pi_{\mathcal{I}} \in \mathit{Pth}$ satisfies the Ltl formula ψ.

Clearly, we want to somehow combine the automaton and the arena because $\mathfrak{P}_{\mathcal{I}}^{\psi}$ represents the winning condition of $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$. However, we cannot simply take their product because, informally, they work at different, albeit constant, "speeds". While $\mathfrak{P}_{\mathcal{I}}^{\psi}$ performs a temporal transition at every step, it takes exactly $|\wp| + 1$ turns before a different underlying global state is reached by $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$. To cater for this asynchrony, we can make the parity automaton "wait" for $|\wp| + 1$ steps before each actual transition.

We do this by extending the state of $\mathfrak{P}^{\psi}_{\mathcal{I}}$ with a simple counter. A path $t_0 t_1 t_2 \cdots \in T^{\omega}$ in $\mathfrak{P}^{\psi}_{\mathcal{I}}$ will then become the following path in the *delayed automaton* $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$:

$$\underbrace{(t_0,0)\,(t_0,1)\ldots(t_0,|\wp|-1)\,(t_0,|\wp|)}_{|\wp|+1 \text{ steps before } \delta \text{ transition}}(t_1,0)\,(t_1,1)\cdots \in (T \times \{0,\ldots,|\wp|\})^{\omega}$$

This idea of adding a counter to the state of the parity automaton is captured in the following definition.

Definition 5.6 (Delayed Automaton)**.** Let $\mathcal{I}$ be an interpreted system, $\mathfrak{P}^{\psi}_{\mathcal{I}} = (T, G, t_I, \delta, c)$ a deterministic parity automaton for an Sl[1g] formula $\psi \in$ *SL[1G]* in $\mathcal{I}$, and $\wp \in$ *QPre* a quantification prefix. We define a deterministic parity automaton $\mathfrak{D}^{\wp\psi}_{\mathcal{I}} \triangleq (T \times \{0,\ldots,|\wp|\}, G, (t_I, 0), \delta_{\mathfrak{D}}, c_{\mathfrak{D}})$, called the *delayed automaton* for ψ over $\wp$ in $\mathcal{I}$, where the transition and colouring function are defined for all $t \in T$, $g \in G$, and $0 \leq k \leq |\wp|$ as:

$$\delta_{\mathfrak{D}}((t,k),g) \triangleq \begin{cases} (t,k+1) & \text{if } k < |\wp| \\ (\delta(t,g),0) & \text{if } k = |\wp| \end{cases} \qquad\qquad c_{\mathfrak{D}}((t,k)) \triangleq c(t)$$

Similarly to the other automata, we shall sometimes refer to a delayed automaton $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ as $\mathfrak{D}^{\varphi}_{\mathcal{I}}$ although it is independent of the binding prefix $\flat$. Informally, the delayed automaton $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ is now working at the "same speed" as the arena $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$. For example, Figure 5.1b shows the delayed automaton $\mathfrak{D}^{\gamma}_{\mathcal{I}}$ for the Sl[1g] principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E},e)(1,x)(2,y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$ in the toy model in Figure 3.1. It was obtained by adding a counter with range 0–3 to the parity automaton $\mathfrak{P}^{\gamma}_{\mathcal{I}}$ in Figure 5.1a. Note that both the delayed automaton $\mathfrak{D}^{\gamma}_{\mathcal{I}}$ and the formula arena $\mathcal{A}^{\gamma}_{\mathcal{I}}$ in Figure 5.1c perform the same number of intermediate steps before a true transition occurs.

Let $(g_0, []) \in G \times Dec^{\wp\flat}_{\mathcal{I}}$ be some state of $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$. If we run both $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ and $\mathfrak{D}^{\wp\flat\psi}_{\mathcal{I}}$ in parallel starting from $(g_0, [])$ and $(t_I, 0)$ respectively, we obtain the following infinite paths:

$$\begin{tikzcd}[column sep=small]
(g_0,[]) \arrow[r] & (g_0,[a_0]) \arrow[r] & \cdots \arrow[r] & \left(g_0,\left[a_0,\ldots a_{|\wp|-1}\right]\right) \arrow[rr, "\text{temporal transition}"'] \arrow[rr, dashed, bend left] \arrow[drr, dashed, bend right=60] & & (t(g_0,a),[]) \arrow[r] & \cdots \\
(t_I,0) \arrow[r] & (t_I,1) \arrow[r] & \cdots \arrow[r] & (t_I,|\wp|) \arrow[rr] & & (\delta(t_I,g_0),0) \arrow[r] & \cdots
\end{tikzcd}$$

where $a \triangleq \left[a_0,\ldots,a_{|\wp|-1}\right]^{Act}$ is the joint action implied by the accumulated decision. We can observe that when a temporal transition occurs, the underlying global state of the arena g_0 is used for two purposes simultaneously (dashed arrows): *(i)* it determines the next state of the arena together with the joint action a and *(ii)* it serves as an input to the parity automaton. Notice that the length of the decision tuple is always equal to the value of the counter. We are now ready to define the *combined parity game* $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$.

Definition 5.7 (Combined Parity Game)**.** Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi \in$ *SL[1G]* an Sl[1g] basic principal formula. Furthermore, let $\mathcal{A}^{\wp\flat}_{\mathcal{I}} = (V_0, V_1, E)$ be the formula arena of $\mathcal{I}$ for $\wp\flat\psi$ with states $V \subseteq G \times Dec^{\wp\flat}_{\mathcal{I}}$ and $\mathfrak{D}^{\wp\psi}_{\mathcal{I}} = (T \times \{0,\ldots,|\wp|\}, G, (t_I,0), \delta_{\mathfrak{D}}, c_{\mathfrak{D}})$ the delayed automaton for ψ over $\wp$ in $\mathcal{I}$. We construct a two-player *combined parity game* $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}} \triangleq ((V_0 \times T, V_1 \times T, E_{\mathfrak{G}}), c_{\mathfrak{G}})$ of $\mathcal{I}$ for $\wp\flat\psi$ where for all states $(g,d,t) \in V \times T$, the colouring function is defined as $c_{\mathfrak{G}}((g,d,t)) \triangleq c_{\mathfrak{D}}((t,|d|)) = c(t)$. The transition relation is defined as follows:

$$E_{\mathfrak{G}} \triangleq \{((g,d,t),(g',d',t')) \in (V \times T) \times (V \times T) \mid E((g,d),(g,d)) \wedge \delta_{\mathfrak{D}}((t,|d|),g)\}$$

For all $g \in G$, we define $\mathsf{start}(g) \triangleq (g, [], t_I)$.

Figure 5.2 shows the combined parity game $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ of the toy model in Figure 3.1 for the Sl[1g] principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E},e)(1,x)(2,y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$ introduced earlier. We obtained $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ by combining the formula arena $\mathcal{A}^{\gamma}_{\mathcal{I}}$ in Figure 5.1c and the delayed automaton $\mathfrak{D}^{\gamma}_{\mathcal{I}}$ in Figure 5.1b. It highlights the winning regions and strategies of the existential player[5], which we will discuss shortly.

Finally, we have everything we need to explain how Sl[1g] model checking can be reduced to solving two-player parity games. This reduction (together with its implementation) is most probably the *biggest achievement* of the whole project.

[5]Note that the existential and universal player are players 0 and 1 of the combined parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$ respectively.

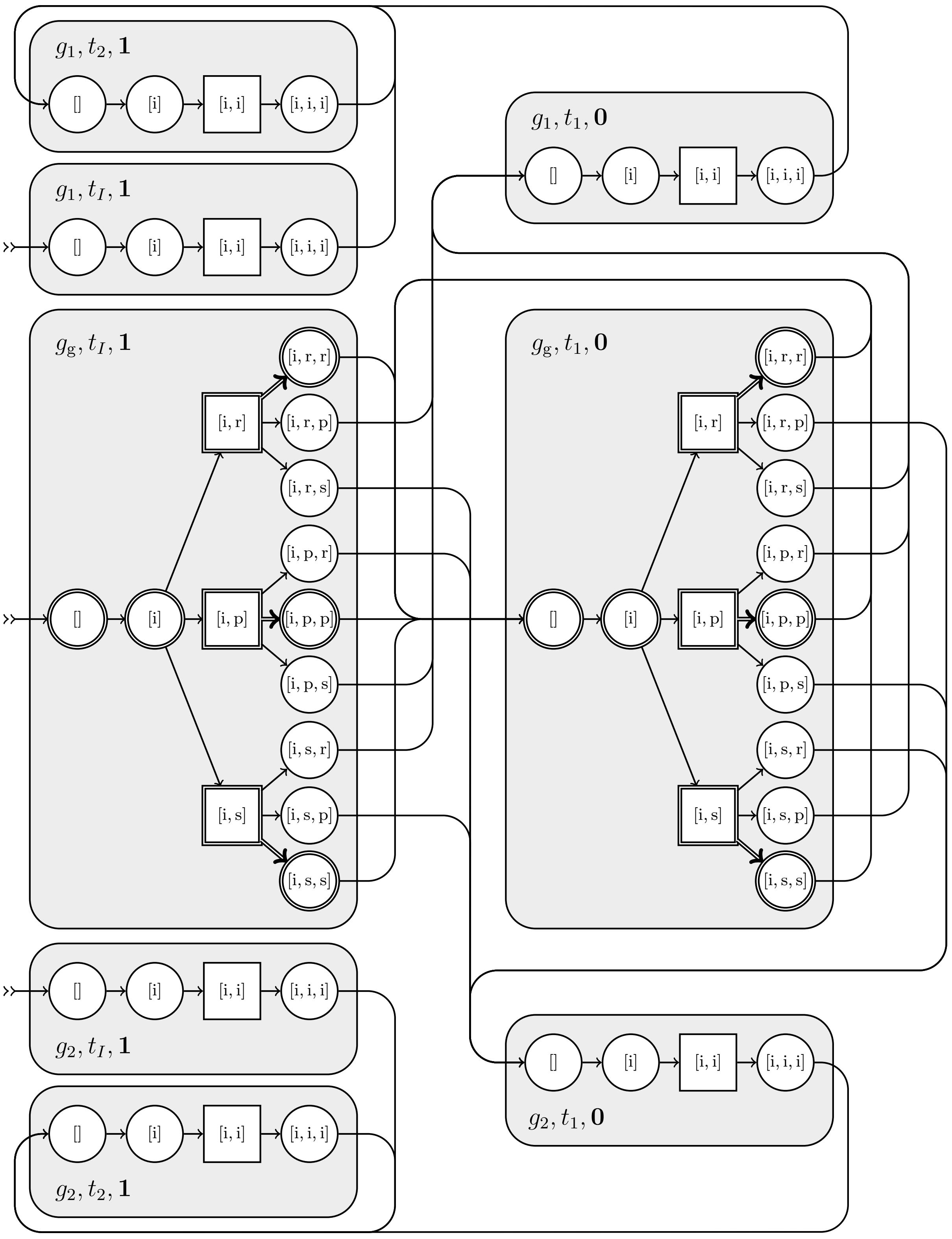


Figure 5.2: Combined parity game $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ of the toy model for the SL[1G] basic principal sentence $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. Existential and universal player states are represented by squares and circles respectively. Winning states and strategies for the existential player have double edges. The universal player can play an arbitrary strategy in his winning region in order to win. The bold numbers (**0** and **1**) refer to the colours assigned by the colouring function. The three states on the left with incoming arrows are $\mathsf{start}(g_1)$, $\mathsf{start}(g_\mathrm{g})$, and $\mathsf{start}(g_2)$ respectively.

Theorem 5.1. Let $\mathcal{I}$ be an interpreted system, $\wp\flat\psi$ an $\textsc{Sl}[1\textsc{g}]$ basic principal sentence, and $g \in G$ an arbitrary global state. Then the following holds:

$$\mathcal{I}, \emptyset, g \models_{\textsc{Sl}} \wp\flat\psi \quad \text{iff} \quad \mathsf{start}(g) \in W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$$

where $W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$ is the winning region of the existential player in the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$.

Proof. The claim follows directly from out construction of the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$. However, given the importance of this result, we feel that a more direct proof should be provided. Essentially, we use the following two facts:

1. The delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ accepts all infinite paths $\tau \in \mathit{Pth}$ such that the infinite path $\pi = \tau(|\wp|)\tau(2\,|\wp|+1)\ldots\tau((k+1)\,|\wp|+k)\cdots \in \mathit{Pth}$ satisfies the $\textsc{Ltl}$ formula ψ, i.e. $\mathcal{I} \models_{\textsc{Ltl}} \psi$. This comes directly from the correctness of the existing techniques (the standard translation, transformation from generalised Büchi automata to Büchi automata, and the determinisation procedure) presented in Section 2.4.

2. The winning condition for a path π in the pseudo-$\textsc{Ltl}$ game $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$, based on the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$, is that the underlying path $\pi_{\mathcal{I}}$ in $\mathcal{I}$ satisfies the $\textsc{Ltl}$ formula ψ. Lemma 5.2 asserts that $(g, [])$ is winning for the existential player in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ iff $\mathcal{I}, \emptyset, g \models_{\textsc{Sl}} \wp\flat\psi$.

We prove both directions of the equivalence separately:

$\Rightarrow$: Assume that $\mathcal{I}, \emptyset, g \models_{\textsc{Sl}} \wp\flat\psi$. Then by Lemma 5.2, $(g, [])$ is winning for the existential player in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$. This means that the existential player can enforce that the infinite path $\pi \in V^\omega$ starting in $(g, [])$ in the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ is such that the underlying path $\pi_{\mathcal{I}} \in \mathit{Pth}$ in $\mathcal{I}$ satisfies the $\textsc{Ltl}$ formula ψ. By correctness of the existing procedures, $\pi_{\mathcal{I}}$ is accepted by $\mathfrak{P}_{\mathcal{I}}^{\psi}$. This means that the infinite path constructed from the first elements of the tuples in π (i.e. a path in which each state from $\pi_{\mathcal{I}}$ occurs $|\wp|+1$ times in a row) is accepted by the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ (by construction). Hence, the existential player can enforce an infinite path in $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ starting in $\mathsf{start}(g)$, which is winning for him (since the winner is decided by the parity condition of $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$). Therefore, $\mathsf{start}(g) \in W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$.

$\Leftarrow$: Assume that $\mathsf{start}(g) \in W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$. By construction of $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$, this means that the existential player can enforce an infinite path $\pi \in V^\omega$ starting in $(g, [])$ in the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$, which is accepted by the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$. Given the shape of π (the first element of the tuple is always repeated $|\wp|+1$ times in a row) and the acceptance condition of $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ explained above, the underlying path $\pi_{\mathcal{I}} \in \mathit{Pth}$ in $\mathcal{I}$ satisfies the $\textsc{Ltl}$ formula ψ. By definition of the winning condition (namely that $\mathcal{I}, \pi_{\mathcal{I}} \models_{\textsc{Ltl}} \psi$), the existential player can enforce the path π in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$ (as it is based on $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$), which is winning for him. Hence $\pi(0) = (g, [])$ is winning for the existential player in $\mathfrak{L}_{\mathcal{I}}^{\wp\flat\psi}$. Finally, by Lemma 5.2, we have $\mathcal{I}, \emptyset, g \models_{\textsc{Sl}} \wp\flat\psi$ as required.

□

Therefore, calculating the winning regions $W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$ and $W_1(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi})$ of the existential and universal player in the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ yields the solution to the model checking problem. This can be done using existing algorithms for solving parity games such as the one in Figure 2.9.

Corollary 5.1. Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi$ an $\textsc{Sl}[1\textsc{g}]$ basic principal sentence. The set of all global states $\|\wp\flat\psi\|_{\mathcal{I}} \subseteq G$ of $\mathcal{I}$ in which $\wp\flat\psi$ holds is $\|\wp\flat\psi\|_{\mathcal{I}} = \left\{ g \in G \,\middle|\, \mathsf{start}(g) \in W_0(\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}) \right\}$.

Consider again the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\gamma}$ in Figure 5.2 of the toy model in Figure 3.1 for the $\textsc{Sl}[1\textsc{g}]$ principal formula $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. We have $\mathsf{start}(g_{\mathrm{g}}) \in W_0(\mathfrak{G}_{\mathcal{I}}^{\gamma})$ but $\mathsf{start}(g_1), \mathsf{start}(g_2) \notin W_0(\mathfrak{G}_{\mathcal{I}}^{\gamma})$. Therefore, by Theorem 5.1, $\mathcal{I}, \emptyset, g_{\mathrm{g}} \models_{\textsc{Sl}} \gamma$, $\mathcal{I}, \emptyset, g_1 \not\models_{\textsc{Sl}} \gamma$, and $\mathcal{I}, \emptyset, g_2 \not\models_{\textsc{Sl}} \gamma$. Hence, we have $\|\gamma\|_{\mathcal{I}} = \{g_{\mathrm{g}}\}$. This result agrees with the intuitive meaning of the formula presented earlier.

We have now covered all steps of our new model checking algorithm for $\textsc{Sl}[1\textsc{g}]$. The complete algorithm is shown in Figure 5.3. It expects an $\textsc{Sl}[1\textsc{g}]$ *principal sentence* (not necessarily basic). As

```
function CheckSl[1g]($\wp\flat\psi$)
    $\mathfrak{A}^{\psi}_{\mathcal{I}}$ := GenBüchi($\psi$)    ▷ Standard translation (Figure 2.6).
    $\mathfrak{B}^{\psi}_{\mathcal{I}}$ := Büchi($\mathfrak{A}^{\psi}_{\mathcal{I}}$)    ▷ Convert to a Büchi automaton (Subsection 2.4.1).
    $\mathfrak{P}^{\psi}_{\mathcal{I}}$ := Parity($\mathfrak{B}^{\psi}_{\mathcal{I}}$)    ▷ Determinise (Subsection 2.4.4).
    $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ := Delayed($\mathfrak{P}^{\psi}_{\mathcal{I}}$, $\wp$)    ▷ Convert to a delayed automaton (Definition 5.6).
    $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ := Arena($\mathcal{I}$, $\wp$, $\flat$)    ▷ Arena construction (Definition 5.4).
    $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$ := Combine($\mathcal{A}^{\wp\flat}_{\mathcal{I}}$, $\mathfrak{D}^{\wp\flat\psi}_{\mathcal{I}}$)    ▷ Combine into a parity game (Definition 5.7).
    $(W_0, W_1)$ := SolveParity($\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$)    ▷ Solve parity game (Figure 2.9).
    return $\{g \in G \mid \mathsf{start}(g) \in W_0\}$    ▷ Return $\|\wp\flat\psi\|_{\mathcal{I}}$ (Corollary 5.1).
end function
```

Figure 5.3: The model checking algorithm for Sl[1g] with references to the relevant parts of this thesis. The argument $\wp\flat\psi$ is an arbitrary Sl[1g] principal sentence. The procedure GenBüchi on line 2 calls CheckSl[1g] recursively on direct principal subformulas of $\wp\flat\psi$ and replaces them with new atoms as explained on page 89. Therefore, $\wp\flat\psi$ can be treated as a *basic* principal sentence from line 3 onwards.

explained earlier, an arbitrary Sl[1g] sentence ψ can be converted to an equivalent principal sentence as follows:

$$[\langle\langle x_i\rangle\rangle]_{i\in Agt}\,[(i, x_i)]_{i\in Agt}\,\psi$$

For example, the Sl[1g] sentence $p_1 \vee p_2$ for the toy model in Figure 3.1 can be converted into an equivalent principal sentence $\langle\langle x_{\mathrm{E}}\rangle\rangle\langle\langle x_1\rangle\rangle\langle\langle x_1\rangle\rangle(\mathrm{E}, x_{\mathrm{E}})(1, x_1)(2, x_2)(p_1 \vee p_2)$, which can then be passed as input to the algorithm in Figure 5.3.

The algorithm is further discussed in the rest of this chapter: Its complexity is discussed in Subsection 5.2.2. Subsection 5.2.3 explains how it can be used for the purposes of strategy synthesis. An efficient symbolic implementation of the algorithm is provided in Subsection 5.2.4. Finally, an optimisation for improving its performance called separate determinisation is presented in Subsection 5.2.5.

5.2.2 Complexity

In this subsection, we will discuss the complexity of the model checking algorithm presented in the previous subsection and compare it with the theoretical complexity of the Sl[1g] model checking problem. Recall that the decision problem is as follows:

> Given an interpreted system $\mathcal{I}$, a global state $g \in G$, and an Sl[1g] formula $\varphi \in$ *SL[1G]*, determine whether $\mathcal{I}, \emptyset, g \models_{\text{Sl[1g]}} \varphi$.

It has been shown that the theoretical complexity of this problem is 2ExpTime-complete with respect to the size of the formula $|\varphi|$ and P-complete with respect to the size of the model $|\mathcal{I}|$. We argue that our new algorithm satisfies both bounds.

Theorem 5.2. Let $\mathcal{I}$ be an interpreted system and $\varphi \in$ *SL[1G]* an Sl[1g] sentence. Our model checking algorithm calculates the set of all global states $\|\varphi\|_{\mathcal{I}} \subseteq G$ satisfying φ in time $|\mathcal{I}|^{2^{O(|\varphi|)}}$.

Proof. We start by considering an arbitrary Sl[1g] *basic principal sentence* $\wp\flat\psi$ and show that our claim holds. The model checking algorithm (see Figure 5.3) proceeds as follows:

1. It constructs a non-deterministic generalised Büchi automaton $\mathfrak{A}^{\psi}_{\mathcal{I}}$ with $O(2^{|\psi|})$ states (see Proposition 2.2).
2. It converts $\mathfrak{A}^{\psi}_{\mathcal{I}}$ to a non-deterministic Büchi automaton $\mathfrak{B}^{\psi}_{\mathcal{I}}$. $\mathfrak{A}^{\psi}_{\mathcal{I}}$ can have up to $O(|\psi|)$ fairness constraints, so the resulting automaton will have $O(|\psi| \times 2^{|\psi|}) = 2^{O(|\psi|)}$ states.
3. It transforms $\mathfrak{B}^{\psi}_{\mathcal{I}}$ to a deterministic parity automaton $\mathfrak{P}^{\psi}_{\mathcal{I}}$ with $2^{\left(2^{O(|\psi|)}+1\right)^2} = 2^{2^{O(|\psi|)}}$ states and $2 \times 2^{O(|\psi|)} = 2^{O(|\psi|)}$ colours (see Proposition 2.5).
4. It transforms $\mathfrak{P}^{\psi}_{\mathcal{I}}$ to the delayed automaton $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ with $|\wp| \times 2^{2^{O(|\psi|)}}$ states.

5. It constructs the arena $\mathcal{A}^{\flat\wp}_{\mathcal{I}}$ with $O(|\mathcal{I}|\times|\wp|)$ states.

6. It combines $\mathcal{A}^{\flat\wp}_{\mathcal{I}}$ and $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ into the combined parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$ with $|\mathcal{I}|\times|\wp|\times 2^{2^{O(|\psi|)}}$ states and $2^{O(|\psi|)}$ colours.

7. Finally, it solves the combined parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$. The algorithm for solving parity games in Figure 2.9 has time complexity $O(m\times(n/d)^d)$ (Proposition 2.10), where m is the number of edges, n the number of vertices, and d the number of colours. Hence, solving $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$ will take time:

$$\underbrace{\left[|\mathcal{I}|\times|\wp|\times 2^{2^{O(|\psi|)}}\right]^2}_{\text{edges}} \times \underbrace{\left[\frac{|\mathcal{I}|\times|\wp|\times 2^{2^{O(|\psi|)}}}{2^{O(|\psi|)}}\right]^{\overbrace{2^{O(|\psi|)}}^{\text{colours}}}}_{\text{vertices/colours}} = |\mathcal{I}|^{2^{O(|\wp\flat\psi|)}}$$

All the construction steps (1–6) of the algorithm fit into the $|\mathcal{I}|^{2^{O(|\wp\flat\psi|)}}$ time bound. Therefore, the time complexity of our model checking algorithm on SL[1G] *basic principal sentences* is $|\mathcal{I}|^{2^{O(|\wp\flat\psi|)}}$.

Let us now come back to φ, which is an arbitrary SL[1G] sentence (not necessarily principal). We will model check φ in a recursive bottom-up manner as explained in Subsection 5.2.1. Hence, we will model-check at most $|\varphi|$ SL[1G] basic principal sentences of size at most $|\varphi|$. Furthermore, if φ is not a principal sentence, then it must be a Boolean combination[6] of some principal sentences, the results of which we can combine using set operations (instead of adding quantifiers and bindings so that $\wp'\flat'\varphi$ would be a principal sentence). Therefore, φ can be checked in time:

$$\overbrace{|\varphi|\times\underbrace{|\mathcal{I}|^{2^{O(|\varphi|)}}}_{\substack{\text{basic}\\\text{principal}\\\text{sentence}}}}^{\text{arbitrary sentence}} = |\mathcal{I}|^{\left(\log_{|\mathcal{I}|}|\varphi|\right)+2^{O(|\varphi|)}} \le |\mathcal{I}|^{\left(\log_{|\mathcal{I}|}|\varphi|\right)\times 2^{O(|\varphi|)}} = |\mathcal{I}|^{2^{\left(\log_2\log_{|\mathcal{I}|}|\varphi|\right)+O(|\varphi|)}} = |\mathcal{I}|^{2^{O(|\varphi|)}} \tag{5.1}$$

Hence, our claim follows. □

This is a very positive result because our algorithm has the same time complexity as the theoretical algorithm for model checking ATL* presented in [23]. However, it should be noted that the proof above is rather "pessimistic" in the sense that it assumes that each principal subsentence has size $|\varphi|$. In fact, a lower worst-case complexity can be derived:

$$|\varphi| + \sum_{\varphi'\in\mathsf{sub}(\varphi)} |\mathcal{I}|^{2^{O(\mathsf{temp}(\varphi'))}}$$

where $\mathsf{sub}(\varphi)$ is the set of all principal subsentences of φ (including φ itself) and $\mathsf{temp}(\varphi')$ is the number of temporal operators in φ' (excluding strict principal subformulas). The reason why we can use $\mathsf{temp}(\varphi')$ instead of $|\varphi'|$ is that a non-deterministic generalised Büchi automaton for an LTL formula with $\mathsf{temp}(\varphi')$ temporal operators has at most $2^{2\times\mathsf{temp}(\varphi')+1}$ states (see Proposition 2.2).

5.2.3 Strategy Synthesis

The model checking algorithm presented in Subsection 5.2.1 decides *whether* a given SL[1G] formula holds in a given model. In addition, we would also like to know *why* that is the case, i.e. what are the strategies that make the formula true. For example, if the formula $[\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E},e)(1,x)(2,y)\mathsf{G}\,[\neg p_1\wedge\neg p_2]$ is true in a given state, we want to know how the strategy assigned to y depends on the strategies assigned to e and x. Conversely, if the formula is false in some state, we want to know the corresponding strategies for the universally quantified variables e and x.

Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi$ an SL[1G] principal sentence. Our algorithm reduces the SL[1G] model checking problem to solving a parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$. We then find its winning regions

[6]This follows from SL[1G] syntax (Definition 5.2) and the fact that $\mathsf{free}(\varphi)=\emptyset$ (if φ combined its direct principal subsentences using temporal operators, we would have $\mathsf{free}(\varphi)=Agt$).

(W_0, W_1) using the existing algorithm for solving parity games presented in Figure 2.9. As explained in Subsection 2.4.5, the algorithm can also be used to construct the winning strategies for both players. By Proposition 2.8, the winning strategies for $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ are *memoryless*, i.e. they are partial functions $w_\sigma : V_\sigma \times T \rightharpoonup V \times T$ such that $\mathrm{dom}(w_\sigma) = W_\sigma \cap (V_\sigma \times T)$ for players $\sigma \in \{0, 1\}$. Since the domains of w_σ and w_σ are disjoint, we can regard them as one partial function $w : V \times T \rightharpoonup V \times T$ with $\mathrm{dom}(w) = W_0 \cap (V_0 \times T) \cup W_1 \cap (V_1 \times T)$ defined as $w \triangleq w_0 \cup w_1$. Intuitively, w assigns a winning move $w(s) \in V \times T$ to every state $s \in V \times T$ that is in the winning region of the player that it belongs to, i.e. either $W_0 \cap (V_0 \times T)$, or $W_1 \cap (V_1 \times T)$.

Clearly, the winning strategies for the game somehow encode the SL[1G] shared strategies, which we want to synthesise. For example, the winning strategy for the existential player in the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\gamma}$ in Figure 5.2 requires that he matches the universal player's move (notice the emphasised transitions of the form $[\mathrm{i}, a] \Rightarrow [\mathrm{i}, a, a]$ with $a \in \{\mathrm{r}, \mathrm{p}, \mathrm{s}\}$). Nevertheless, there are important differences between the two types of strategies:

1. The *game strategies* are memoryless, partial, and map game states to game states ($w_\sigma : V_\sigma \times T \rightharpoonup V \times T$ with $\sigma \in \{0, 1\}$). There are always exactly two of them, one for the existential player w_0 and one for the universal player w_1.

2. The *agent strategies* are memoryful, total, and map non-empty finite sequences of global states (tracks) to actions ($f_k : \mathit{Trk} \to \mathit{Act}_{\mathsf{sharing}(\flat\psi, \wp(k))}$ with $0 \leq k < |\wp|$). There is one for each quantifier in the quantifier prefix $\wp$.

Let us start with the memory. Could the agent strategies also always be memoryless? No, unfortunately not. To see why, have a look at the example in Subsection 4.1.5, where a very simple SLK formula (which is also an SL[1G] formula) cannot be satisfied using memoryless strategies. Intuitively, the difference is that the game already "contains" the deterministic parity automaton $\mathfrak{P}_{\mathcal{I}}^{\psi}$, which defines the winning condition. The players can thus use its underlying state to select their moves "for free" because it is a part of the arena. This means that if the agents simulate the progress of $\mathfrak{P}_{\mathcal{I}}^{\psi}$ in their memory, they will have almost the same information as they have in $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$. However, they might also need to remember some of the other agents' actions, which are part of the game state space as well.

The issue of w_σ being a partial function is a rather cosmetic one. If it is undefined in some state $s \in V_\sigma \times T$, i.e. $s \notin \mathrm{dom}(w_j)$, it just means that player σ cannot enforce a victory from s. In that case, we can assign an arbitrary move $s' \in V \times T$ such that $E_{\mathfrak{G}}(s, s')$ to player σ in state s. We shall denote this a *total game strategy* $\hat{w}_\sigma$.

Definition 5.8 (Total Strategy)**.** Let $\mathfrak{G}$ be a memoryless determined game with an underlying arena $\mathcal{A} = (V_0, V_1, E)$ with no dead ends, i.e. E is serial. Furthermore, let the winning strategies of $\mathfrak{G}$ be (w_0, w_1). Then a *total strategy* for player $\sigma \in \{0, 1\}$ in $\mathfrak{G}$ is a total function $\hat{w}_\sigma : V_\sigma \to V$ such that for all $v \in V_\sigma$, we have $E(v, \hat{w}_\sigma(v))$ and if $v \in \mathrm{dom}(w_\sigma)$, then $\hat{w}_\sigma(v) = w_\sigma(v)$. The *total solution* of $\mathfrak{G}$ is a total function $\hat{w} : V \to V$ such that $\hat{w} \triangleq \hat{w}_0 \cup \hat{w}_1$.

The third difference is that the output of the game strategies are game states, whereas the agent strategies return actions. Let us have a closer look at what the input an output of $\hat{w}$ look like when selecting an action for the quantifier $\wp(k)$ with $0 \leq k < |\wp|$:

$$\hat{w}((g, [a_0, \ldots, a_{k-1}], t)) = (g, [a_0, \ldots, a_{k-1}, a_k], t)$$

It merely appends the action $a_k \in \mathit{Act}_{\mathsf{sharing}(\flat\psi, \wp(k))}$, which the strategy $\wp(k)$ will do in the current state $g \in G$, to the tuple of actions $[a_0, \ldots, a_{k-1}]$ of the strategy variables it depends on $(\wp(0), \ldots, \wp(k-1))$. Intuitively, the equality above implies that the next action which the strategy assigned to the variable $\wp(k)$ selects will depend only on the global state $g \in G$, the underlying parity automaton state $t \in T$, and the next actions selected by the preceding strategies in the quantifier prefix. Recall that the underlying global and parity automaton states g and t in the equation above change only when a temporal transition occurs (i.e. every $|\wp| + 1$ steps in the game). Moreover, the next parity automaton state $t' = \delta(t, g)$ does not depend on the accumulated decisions $\left[a_0, \ldots, a_{|\wp|-1}\right]$ of the players. Hence, if we regard the states of the parity automaton T as a memory, we get the following *memory update* and *next action* functions for variable $\wp(k)$ with $0 \leq k < |\wp|$:

$$\delta_k : T \times G \to T \qquad\qquad \alpha_k : T \times G \times \underbrace{\left[\prod_{l=0}^{k-1} Act_{\mathsf{sharing}(\flat\psi, \wp(l))}\right]}_{\substack{\text{next actions of strategies}\\ \wp(0), \ldots, \wp(k-1)}} \to \underbrace{Act_{\mathsf{sharing}(\flat\psi, \wp(k))}}_{\substack{\text{next action of}\\ \text{strategy } \wp(k)}}$$

This is very similar to a finite memory strategy (Definition 2.56). However, α_k depends on the next actions of other strategies in the current state. Therefore, strictly speaking, the strategy might require an infinite amount of memory. To see why this is the case, consider again the SL[1G] formula for the toy model in Figure 3.1 $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle \mathsf{G}\, \neg\,(p_1 \vee p_2)$. It clearly holds in the initial state g_{g}: Player 2 just always plays the same action as player 1. Hence, for all strategies x for player 1, there exists a corresponding strategy y for player 2, namely $y = x$, which ensures this property. However, consider the case when player 1 plays rock when the current round number is a square and scissors otherwise:

$$\begin{array}{cccccccccccc} 0 & 1 & & & 4 & & & & & 9 & & \\ \mathrm{r} & \mathrm{r} & \mathrm{s} & \mathrm{s} & \mathrm{r} & \mathrm{s} & \mathrm{s} & \mathrm{s} & \mathrm{s} & \mathrm{r} & \mathrm{s} & \ldots \end{array}$$

Clearly, player 2 can satisfy the property by applying the exact same strategy. However, such a strategy is not finite memory because it must somehow keep track of the number of rounds until the next square, which can become arbitrarily large. Nevertheless, the strategy can be regarded as having finite memory in some sense if we include the next actions of other strategies in the input to the next action function (exactly like we did for α_k).

To sum up, the next action $f_k(\pi)$ of the synthesised strategy for variable $\wp(k)$ depends on: *(i)* the track through the interpreted system so far $\pi \in \mathit{Trk}$ and *(ii)* the next actions $f_0(\pi), \ldots, f_{k-1}(\pi)$ of the strategies for variables $\wp(0), \ldots, \wp(k-1)$. The following definition expresses this idea formally.

Definition 5.9 (SL[1G] Strategy Synthesis)**.** Let $\mathcal{I}$ be an interpreted system, $\wp\flat\psi$ an SL[1G] principal sentence, $0 \leq k < |\wp|$ an integer, and $f_0, \ldots, f_{k-1}$ shared strategies for variables $\wp(0), \ldots, \wp(k-1)$. Furthermore, let $\mathfrak{P}^{\psi}_{\mathcal{I}} = (T, G, t_I, \delta, c)$ be the deterministic parity automaton for ψ and (w_0, w_1) the winning strategies of the combined parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$. Then we define the strategy $f_k : \mathit{Trk} \to Act_{\mathsf{sharing}(\flat\psi, \wp(k))}$ for variable $\wp(k)$ for all tracks $\pi \in \mathit{Trk}$ implicitly as:

$$\hat{w}((\mathsf{last}(\pi), [f_0(\pi), \ldots, f_{k-1}(\pi)]\,, \delta(t_I, \pi_{\leq|\pi|-2}))) = (\mathsf{last}(\pi), [f_0(\pi), \ldots, f_{k-1}(\pi), \underline{f_k(\pi)}], \delta(t_I, \pi_{\leq|\pi|-2}))$$

where $\delta(t_I, \pi_{\leq|\pi|-2}) \triangleq \delta(\ldots\delta(\delta(t_I, \pi(0)), \pi(1))\ldots, \pi(|\pi|-2))$.

While the definition above might look very complex, it is simply reusing the winning strategies of the combined parity game $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$. Once more, consider the combined parity game $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ in Figure 5.2 of the toy model in Figure 3.1 for the formula $\gamma \triangleq [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. Assume that the history of the game so far is $\pi = g_{\mathrm{g}} g_{\mathrm{g}} g_{\mathrm{g}}$. Hence, the current global state is $\mathsf{last}(\pi) = \pi(2) = g_{\mathrm{g}}$. As the environment has only one available action in g_{g}, we know that $f_0(\pi) = \mathrm{i}$. Let us further assume that player 1 has played scissors, i.e. $f_1(\pi) = \mathrm{s}$. The decision accumulated so far is thus $[f_0(\pi), f_1(\pi)] = [\mathrm{i}, \mathrm{s}]$ and the state of the parity automaton is $\delta(t_I, \pi) = \delta(\delta(t_I, g_{\mathrm{g}}), g_{\mathrm{g}}) = t_1$. The current implied state of the game $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ in Figure 5.2 is therefore $(g_{\mathrm{g}}, [\mathrm{i}, \mathrm{s}]\,, t_1)$. We can see that this state is winning for the existential player and we have $w_0((g_{\mathrm{g}}, [\mathrm{i}, \mathrm{s}]\,, t_1)) = (g_{\mathrm{g}}, [\mathrm{i}, \mathrm{s}, \underline{\mathrm{s}}]\,, t_1)$. Hence, $f_2(\pi) = \mathrm{s}$, i.e. player 2 should also play scissors. This agrees with our expectation.

We shall now state the relationship between SL[1G] model checking and strategy synthesis. Informally, if an SL[1G] principal formula $\wp\flat\psi$ holds in a global state $g \in G$ and we are given arbitrary strategies for the universally quantified variables in $\wp$, the strategies for the existentially quantified variables synthesised[7] according to Definition 5.9 ensure that the formula ψ holds in g.

Theorem 5.3. Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi$ an SL[1G] principal formula. Furthermore, for each index $0 \leq k < |\wp|$ (in increasing order), let f_k be *(i)* an arbitrary shared strategy $f_k \in$

[7]Note that although we assume that we are given strategies for all universally quantified variables up front (before we synthesise the existential ones), Definition 5.9 ensures that the synthesised strategy for a variable $\wp(i)$ does not depend on the strategy for a variable $\wp(j)$ with $i < j$.

$SStr_{\mathsf{sharing}(\flat\psi,\wp(k))}$ if $\wp(k)$ is a *universal* quantifier or *(ii)* a strategy synthesised according to Definition 5.9 if $\wp(k)$ is an *existential* quantifier. Then for all global states $g \in G$:

$$\text{if} \quad \mathcal{I}, \emptyset, g \models_{\text{SL}} \wp\flat\psi \quad \text{then} \quad \mathcal{I}, \chi, g \models_{\text{SL}} \psi$$

where $\chi \in CAsg$ is a complete assignment with $\text{dom}(\chi) = Agt$ such that for all indices $0 \leq k < |\wp|$ and agents $i \in \mathsf{sharing}(\flat\psi, \wp(k))$ we have $\chi(i) \triangleq f_k$.

Proof. Let $\mathfrak{P}_{\mathcal{I}}^{\psi} = (T, G, t_I, \delta, c)$ be the deterministic parity automaton equivalent to the LTL formula ψ and $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi} = ((V_0 \times T, V_1 \times T, E_{\mathfrak{G}}), c_{\mathfrak{G}})$ the combined parity game in $\mathcal{I}$ for $\wp\flat\psi$ with winning regions (W_0, W_1) and strategies (w_0, w_1). Note that we can treat ψ as an LTL formula because all direct principal subsentences are removed even before determinisation (as explained in Figure 5.3). We shall now prove the implication.

Take an arbitrary state $g \in G$ and assume that $\mathcal{I}, \emptyset, g \models_{\text{SL}} \wp\flat\psi$. Let $\pi = \mathsf{play}(\chi, g)$ be the infinite play of the game. We want to show that the LTL formula ψ holds along π. By the definition of a play (Definition 2.26) and the structure of the assignment χ, we have $\pi(0) = g$ and for all $i \geq 0$, $\pi(i+1) = t\left(\pi(i), \left[f_0(\pi_{\leq i}), \ldots, f_{|\wp|-1}(\pi_{\leq i})\right]^{Act}\right)$. The LTL formula ψ holds along π iff $\mathfrak{P}_{\mathcal{I}}^{\psi}$ accepts π. By construction of the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ (Definition 5.7) and the fact that $\left[f_0(\pi_{\leq i}), \ldots, f_{|\wp|-1}(\pi_{\leq i})\right]$ is a valid decision, π is accepted by $\mathfrak{P}_{\mathcal{I}}^{\psi}$ iff the following path is winning for the existential player in the game:

$$\begin{aligned}\tau \triangleq{}& (\pi(0), [], t_I)\, (\pi(0), [f_0(\pi_{\leq 0})], t_I) \ldots \\ & \left(\pi(0), [f_0(\pi_{\leq 0}), \ldots, f_{|\wp|-1}(\pi_{\leq 0})], t_I\right) (\pi(1), [], \delta(t_I, \pi(0))) \ldots\end{aligned}$$

We will now argue that $\tau \in (V \times T)^{\omega}$ must be winning for the existential player in $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$.

Firstly, we show that τ always stays within the winning region W_0 of the existential player. Assume (for contradiction) that this is not the case, i.e. that there is some least position $k \geq 0$ such that $\tau(k) \notin W_0$. By Theorem 5.1, $\tau(0) = (g, [], t_I) \in W_0$ so $k > 0$. Consider the preceding state $\tau(k-1)$. If $\tau(k-1)$ belongs to the universal player, it would be in his winning region W_1 (as he can clearly move to another winning state $\tau(k) \in W_1 = V \setminus W_0$). This is in contradiction with the assumption that k is the first index such that $\tau(k) \notin W_0$. So $\tau(k-1)$ must be a state which belongs to the existential player. This state must be of the form $\tau(k-1) = (\pi(j), [f_0(\pi_{\leq j}), \ldots, f_{l-1}(\pi_{\leq j})], \delta(t_I, \pi_{\leq j-1}))$ for some $j \geq 0$ and $0 \leq l < |\wp|$ such that $\wp(l)$ is an existential quantifier. Since $\tau(k-1) \in V_0 \cap W_0$, we must have $\tau(k-1) \in \text{dom}(w_0)$, i.e. the winning game strategy is defined for $\tau(k-1)$. Moreover, by Definition 5.9, we have $w_0(\tau(k-1)) = (\pi(j), [f_0(\pi_{\leq j}), \ldots, f_l(\pi_{\leq j})], \delta(t_I, \pi_{\leq j-1})) = \tau(k)$. But then $\tau(k)$ must be a winning state for the existential player. Therefore, by contradiction, τ stays always in the winning region of the existential player W_0.

Secondly, we show that for all $k \geq 0$, if $\tau(k) \in V_0$, then $\tau(k+1) = w_0(\tau(k))$, i.e. the path τ follows the winning strategy. Take an arbitrary index $k \geq 0$ such that $\tau(k) \in V_0$. This state must be of the form $\tau(k) = (\pi(j), [f_0(\pi_{\leq j}), \ldots, f_{l-1}(\pi_{\leq j})], \delta(t_I, \pi_{\leq j-1}))$ for some $j \geq 0$ and $0 \leq l < |\wp|$ such that $\wp(l)$ is an existential quantifier. By Definition 5.9, we have $w_0(\tau(k)) = (\pi(j), [f_0(\pi_{\leq j}), \ldots, f_l(\pi_{\leq j})], \delta(t_I, \pi_{\leq j-1})) = \tau(k+1)$ as required.

We have just shown that τ always stays within the winning region of the existential player W_0 and follows the winning strategy w_0. Therefore, it is a winning path for the existential player in $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$. This implies that $\mathfrak{P}_{\mathcal{I}}^{\psi}$ accepts π. Hence, the LTL formula ψ is true along π. Our claim follows. □

This approach can also be used to synthesise universally quantified strategies when a SL[1G] principal formula $\wp\flat\psi$ does *not hold* in a state $g \in G$ of an interpreted system $\mathcal{I}$, i.e. when $\mathcal{I}, \emptyset, g \not\models_{\text{SL}} \wp\flat\psi$. Since we have defined interpreted systems on transition *functions* (see Definition 2.5), by SL semantics (see Definition 2.29), this is equivalent to $\mathcal{I}, \emptyset, g \models_{\text{SL}} \overline{\wp}\flat\neg\psi$, where $\overline{\wp}$ is equal to $\wp$ with swapped quantifiers (e.g. for $\wp = [\![e]\!][\![x]\!]\langle\!\langle y\rangle\!\rangle$ we have $\overline{\wp} = \langle\!\langle e\rangle\!\rangle\langle\!\langle x\rangle\!\rangle[\![y]\!]$). We can thus synthesise universally quantified strategies in $\wp\flat\psi$ by synthesising existentially quantified strategies in $\overline{\wp}\flat\neg\psi$ (using Theorem 5.3). Observe that this is not true in general when using transition *relations* as we might have both $\mathcal{I}, \emptyset, g \not\models_{\text{SL}} \wp\flat\psi$ and $\mathcal{I}, \emptyset, g \not\models_{\text{SL}} \overline{\wp}\flat\neg\psi$. To see how this can happen, consider the case when the LTL formula is $\psi = \mathsf{X}\, p$ and the transition relation is completely non-deterministic. Then no existential (universal) strategies can ensure p will be true (false) in the next state. One way to tackle this issue is to quantify the non-determinism explicitly, i.e. set $\wp' = \wp\,[\![n]\!]$ and $\flat' = \flat\,(\mathrm{N}, n)$. Essentially, we transform the transition relation into an agent with a universally quantified strategy and a transition function. Then $\mathcal{I}, \emptyset, g \not\models_{\mathrm{SL}} \wp'\flat'\psi$ implies $\mathcal{I}, \emptyset, g \models_{\mathrm{SL}} \overline{\wp'}\flat'\neg\psi$ again so we can perform strategy synthesis. Informally, in doing so, we are assuming that the nondeterminism is "playing on the universal quantifiers' side".

Since the SL[1G] strategy synthesis merely reuses the result generated by the model checking algorithm, namely the winning strategies (w_0, w_1) of the existential and universal player in the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$, it has the same complexity.

Note that SL[1G] strategy synthesis performs essentially the same task as the elementary dependence map $\theta_{\mathrm{SL[1G]}}$ for a quantification prefix over strategies mentioned in Subsection 5.2.1. Similarly to SLK, it is possible to define SL[1G] witness and counterexample strategies (see Definition 4.13). We do not discuss this in detail because the SL[1G] strategy synthesis algorithm presented in this subsection is more general and can handle both concepts without any modifications.

5.2.4 Symbolic Implementation

In this subsection we discuss how the model checking algorithm (and therfore also strategy synthesis) presented in Subsection 5.2.1 can be implemented symbolically using BDDs (see Subsection 2.3.1). This subsection requires a good understanding of symbolically represented interpreted systems. Most importantly, it assumes that the reader knows how global states and joint actions can be represented using Boolean vectors. Please refer to Subsections 2.3.2, 3.3.2, and 4.2.4 for more details about this topic.

Delayed Automaton

We first explain how the delayed automaton (see Definition 5.6) is constructed symbolically as it is a relatively straightforward process. Let $\mathcal{I}$ be an interpreted system and $\wp\flat\psi \in$ *SL[1G]* an SL[1G] principal sentence. Furthermore, assume that the Boolean vectors for representing current global states, next global states, and joint actions are $\overline{v} = (v_0, \ldots, v_{N-1})$, $\overline{v'} = \left(v'_0, \ldots, v'_{N-1}\right)$, and $\overline{w} = (w_0, \ldots, w_{M-1})$. The delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ is constructed as follows (lines 2–5 in the algorithm in Figure 5.3):

1. **Construction of a non-deterministic generalised Büchi automaton $\mathfrak{A}_{\mathcal{I}}^{\psi}$.** The standard translation, which we use for this purpose, is already defined symbolically in Figure 2.6. It returns an automaton formula $\mathfrak{A}_{\mathcal{I}}^{\psi} = \mathcal{A}_{\exists}(\overline{q}, \overline{v}, \mathcal{I}_G, \mathcal{R}_G, \mathcal{F}_G)$ with a Boolean vector of new variables $\overline{q} = (q_0, \ldots, q_{K-1})$ and their next versions $\overline{q'} = \left(q'_0, \ldots, q'_{K-1}\right)$. As explained on page 5.2.1, recursive solution of direct principal subformulas of $\wp\flat\psi$ is performed at this stage.

   Note that we define already the non-generalised Büchi automaton on global states. Therefore, the formula representing an atom $p \in AP$ is $\bigvee_{g \in h(p)} g(\overline{v})$. In the case of the new atoms $p_{\varphi'}$ representing direct principal subsentences, this becomes $\bigvee_{g \in \|\varphi'\|_{\mathcal{I}}} g(\overline{v})$.

2. **Conversion to a non-deterministic Büchi automaton $\mathfrak{B}_{\mathcal{I}}^{\psi}$.** This is a very straightforward procedure. We introduce $L = \lceil \log_2(|\mathcal{F}_G| + 1) \rceil$ Boolean variables $\overline{r} = (r_0, \ldots, r_{L-1})$ (together with their next versions $\overline{r'}$) to represent the counter. The result is an automaton formula $\mathfrak{B}_{\mathcal{I}}^{\psi} = \mathcal{A}_{\exists}(\overline{qr}, \overline{v}, \mathcal{I}_B, \mathcal{R}_B, \mathcal{F}_B)$ such that:

$$\begin{aligned} \mathcal{I}_B &= \mathcal{I}_G \wedge 0(\overline{r}) \\ \mathcal{R}_B &= \mathcal{R}_G \wedge \left(\left[\bigvee_{i=0}^{|\mathcal{F}_G|-1} i(\overline{r}) \wedge \left(\mathcal{F}_G(i) \wedge (i+1)(\overline{r'}) \vee \neg\mathcal{F}_G(i) \wedge i(\overline{r'}) \right) \right] \vee \left[|\mathcal{F}_G|\,(\overline{r}) \wedge 0(\overline{r'}) \right] \right) \\ \mathcal{F}_B &= |\mathcal{F}_G|\,(\overline{r}) \end{aligned}$$

   where $i(\overline{r})$ and $i(\overline{r'})$ are the binary representation of $0 \le i \le |\mathcal{F}_G|$ using the Boolean vectors $\overline{r}$ and $\overline{r'}$ respectively.

3. **Determinisation to a parity automaton $\mathfrak{P}_{\mathcal{I}}^{\psi}$.** A symbolic implementation of the determinisation procedure is provided at the end of Subsection 2.4.4. The result is an automaton formula $\mathfrak{P}_{\mathcal{I}}^{\psi} = \mathcal{A}_{\exists}\left(\overline{t}, \overline{v}, \mathcal{I}_P, \mathcal{R}_P, \mathcal{C}\right)$ with a Boolean vector of new variables $\overline{t} = (t_0, \ldots, t_{P-1})$ (and their next versions).

Note that the Boolean vectors $\overline{q}$ and $\overline{r}$ are not used from the next step onwards.

4. **Transformation to a delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$.** This is again a very straightforward procedure. We introduce $D = \lceil \log_2 (|\wp| + 1) \rceil$ Boolean variables $\overline{d} = (d_0, \dots, d_{D-1})$ (together with their next versions $\overline{d'}$) to represent the counter. The result is an automaton formula $\mathfrak{D}_{\mathcal{I}}^{\wp\psi} = \mathcal{A}_{\exists}\big(\overline{t}\overline{d}, \overline{v}, \mathcal{I}_D, \mathcal{R}_D, \mathcal{C}\big)$ where:

$$\begin{aligned} \mathcal{I}_D &= \mathcal{I}_P \wedge 0(\overline{d}) \\ \mathcal{R}_D &= \mathcal{R}_P \wedge |\wp|\,(\overline{d}) \wedge 0(\overline{d'}) \vee \left[\bigvee_{i=0}^{|\wp|-1} i(\overline{d}) \wedge (i+1)(\overline{d'}) \right] \wedge \left[\bigwedge_{j=0}^{P-1} t_j \leftrightarrow t'_j \right] \end{aligned}$$

We shall now discuss how the formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ can be constructed symbolically and then combined with the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\flat\wp}$ into the parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$.

Formula Arena

As we have stated in Subsection 5.2.1, the formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat} = (V_0, V_1, E)$ (see Definition 5.4) is one of the key concepts of our new model checking algorithm for SL[1G]. Providing a thorough explanation of its symbolic representation is therefore of utmost importance.

We shall first explain how the state space of the arena can be represented symbolically. Recall that the states of $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ are pairs $(g, d) \in V$ where $g \in G$ is a global state and $d \in Dec_{\mathcal{I}}^{\wp\flat}$ a decision. Since the symbolic representation of global states was already discussed in Subsection 4.2.4, we will focus on encoding the decisions. According to Definition 5.4, the decision d is a tuple of actions such that:

1. The length of d is $0 \leq |d| \leq |\wp|$. As we have pointed out in Subsection 5.2.1, when combining the formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ and the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ into the parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$, the length of the decision is always equal to the value of the counter. Therefore, we can use the Boolean variables $\overline{d}$ for the counter of the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ to encode the length of the decision d.

2. For each index $0 \leq k < |d|$, we have $d(k) \in Act_{\mathsf{sharing}(\flat\top, \wp(k))}$, i.e. $d(k)$ must be some action that all agents sharing the variable $\wp(k)$ can perform. We can encode this by introducing $E = \sum_{k=0}^{|\wp|-1} \left\lceil \log_2 \left| Act_{\mathsf{sharing}(\flat\top, \wp(k))} \right| \right\rceil$ Boolean variables $\overline{e} = (e_0, \dots, e_{E-1})$. We will use $\overline{e^k}$ to denote the Boolean variables representing the action assigned to $\wp(k)$ for $0 \leq k \leq |\wp|$. While it is not strictly necessary, we will set the value of $\overline{e^k}$ to zero for $|d| \leq k < |\wp|$ in order to reduce the set of reachable states.

Therefore, an arbitrary state of the arena $(g, d) \in V$ is represented symbolically as follows:

$$(g, d)\,(\overline{v}, \overline{d}, \overline{e}) = g(\overline{v}) \wedge |d|\,(\overline{d}) \wedge \left[\bigwedge_{k=0}^{|d|-1} d(k)(\overline{e^k}) \right] \wedge \left[\bigwedge_{k=|d|}^{|\wp|-1} 0(\overline{e^k}) \right]$$

Consider the arena $\mathcal{A}_{\mathcal{I}}^{\gamma}$ in Figure 5.1c of the toy model in Figure 3.1 for the SL[1G] basic principal sentence $\gamma = [\![e]\!][\![x]\!]\langle\!\langle y \rangle\!\rangle(\mathrm{E}, e)(1, x)(2, y)\mathsf{G}\,[\neg p_1 \wedge \neg p_2]$. The global states of $\mathcal{I}$ are represented using the Boolean vector $\overline{v} = (v_0, v_1)$ (see Subsection 3.3.2). Since $|\wp| = 3$, we need $D = \lceil \log_2 (3+1) \rceil = 2$ Boolean variables $\overline{d} = (d_0, d_1)$ to represent the length of the decision. Finally, we need $E = \lceil \log_2 |\{\mathrm{i}\}| \rceil + 2\,\lceil \log_2 |\{\mathrm{r}, \mathrm{p}, \mathrm{s}, \mathrm{i}\}| \rceil = 4$ Boolean variables $\overline{e} = (e_0, e_1, e_2, e_3)$ to represent the decision. Note that $\overline{e^0} = ()$, $\overline{e^1} = (e_0, e_1)$, and $\overline{e^2} = (e_2, e_3)$. For example, the state $(g, d) = (g_{\mathrm{g}}, [\mathrm{i}, \mathrm{p}])$ in $\mathcal{A}_{\mathcal{I}}^{\gamma}$ is encoded as:

$$(g, d)\,(\overline{v}, \overline{d}, \overline{e}) = \underbrace{\neg v_0 \wedge \neg v_1}_{g = g_{\mathrm{g}}} \wedge \underbrace{\neg d_0 \wedge d_1}_{|d| = 2} \wedge \underbrace{e_0 \wedge \neg e_1}_{d(1) = \mathrm{p}} \wedge \underbrace{\neg e_2 \wedge \neg e_3}_{d(2) = \bot}$$

We are now ready to discuss the symbolic representation of the edge relation $E \subseteq V \times V$ of the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$. There are two cases we have to consider:

1. An action a is *appended* to the decision d in an arena state (g, d). Although this is relatively straightforward, we must make sure that all agents can actually perform the action[8], i.e. $a \in \bigcap_{i \in \mathsf{sharing}(\flat\top, \wp(|d|))} P_i(l_{i\mathrm{E}}(g))$. We can do this by reusing the symbolic implementation of the individual protocols $P_i(\overline{v}, \overline{w})$ (see Subsection 3.3.2). For each $0 \leq k < |\wp|$ this case is encoded as:

$$\begin{aligned} E_1^k(\overline{v}, \overline{d}, \overline{e}, \overline{v'}, \overline{d'}, \overline{e'}) = {} & k(\overline{d}) \wedge (k+1)(\overline{d'}) \wedge \\ & 0(\overline{e^k}) \wedge \left[\bigwedge_{l=k+1}^{|\wp|} 0(\overline{e^l}) \wedge 0(\overline{e'^l}) \right] \wedge \left[\bigwedge_{j=0}^{k-1} \bigwedge_{m=0}^{|\overline{e^j}|-1} e^j_m \leftrightarrow e'^j_m \right] \wedge \\ & \left[\bigvee_{a \in Act_{S_k}} \bigwedge_{i \in S_k} \exists \overline{w_i}.\, P_i(\overline{v_{i\mathrm{E}}}, \overline{w_i}) \wedge a(\overline{w_i}) \wedge a(\overline{e'^k}) \right] \end{aligned}$$

where $S_k = \mathsf{sharing}(\flat\top, \wp(k))$. The first line increments the length of the decision tuple. The second line asserts that the decision does not change arbitrarily. The third line ensures that only possible actions are appended.

2. The decision is complete and thus a *temporal transition* occurs. While this case is slightly more complicated, we can simplify it by reusing the global evolution function $t(\overline{v}, \overline{w}, \overline{v'})$ (see Subsection 3.3.2). Intuitively, we force all agents to perform the action that the decision assigns to them. This case is encoded as:

$$\begin{aligned} E_2(\overline{v}, \overline{d}, \overline{e}, \overline{v'}, \overline{d'}, \overline{e'}) = {} & |\wp|\,(\overline{d}) \wedge 0(\overline{d'}) \wedge \left[\bigwedge_{j=0}^{|\wp|-1} 0(\overline{e'^j}) \right] \wedge \\ & \left[\bigvee_{a \in Act} \exists \overline{w}. a(\overline{w}) \wedge t(\overline{v}, \overline{w}, \overline{v'}) \wedge \bigwedge_{k=0}^{|\wp|-1} \bigwedge_{i \in S_k} a_i(a)(\overline{e^k}) \right] \end{aligned}$$

where $S_k = \mathsf{sharing}(\flat\top, \wp(k))$ for $0 \leq k < |\wp|$. Again, the first line ensures that the decision (and its successor) have the correct form. The second line finds a joint action $a \in Act$ which agrees with the decision and performs the temporal transition.

Finally, the *edge relation* E of the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ is implemented as a disjunction of the two cases above:

$$E(\overline{v}, \overline{d}, \overline{e}, \overline{v'}, \overline{d'}, \overline{e'}) = \left[\bigvee_{k=0}^{|\wp|-1} E_1^k(\overline{v}, \overline{d}, \overline{e}, \overline{v'}, \overline{d'}, \overline{e'}) \right] \vee E_2(\overline{v}, \overline{d}, \overline{e}, \overline{v'}, \overline{d'}, \overline{e'})$$

We are only interested in paths in $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ starting in states $(g, [])$ where $g \in G$ is a reachable global state. A symbolic representation of all *vertices* V of the arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ can thus be obtained as the least fixed point $V = \mathrm{lfp}_Q\,[\{(g, []) \mid g \in G\} \cup \mathsf{suc}_\exists(Q)]$ (similarly to the calculation of the set of reachable states at the end of Subsection 3.3.2):

$$V(\overline{v}, \overline{d}, \overline{e}) = \mathrm{lfp}_\Theta \left[\left(\bigvee_{g \in G} g(\overline{v}) \wedge 0(\overline{d}) \wedge 0(\overline{e}) \right) \vee \exists \overline{v' d' e'}.\, \big(\Theta' \wedge E(\overline{v'}, \overline{d'}, \overline{e'}, \overline{v}, \overline{d}, \overline{e})\big) \right]$$

where Θ' is a Boolean formula equal to Θ with variables in $\overline{v}$, $\overline{d}$, $\overline{e}$ and $\overline{v'}$, $\overline{d'}$, $\overline{e'}$ swapped. The vertices

[8]If we do not ensure this, the game might end up in a dead end $(g, d_\perp)$ when a temporal transition should occur as the accumulated decision $d_\perp$ might not represent a possible joint action in g. By our construction of $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$, $(g, d_\perp)$ would be a dead end for the universal player. Hence, it would always be a winning state for the existential player. Informally, this would "encourage" the existential player to choose actions which cannot be performed in the current state.

of the *existential and universal player* in $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ can in turn be calculated symbolically as follows:

$$V_0(\overline{v},\overline{d},\overline{e}) = V(\overline{v},\overline{d},\overline{e}) \wedge \bigvee_{k=0}^{|\wp|-1} k(\overline{d}) \wedge \mathsf{existential}(k)$$

$$V_1(\overline{v},\overline{d},\overline{e}) = V(\overline{v},\overline{d},\overline{e}) \wedge \bigvee_{k=0}^{|\wp|-1} k(\overline{d}) \wedge \neg\mathsf{existential}(k)$$

where $\mathsf{existstential}(k)$ is true iff $\wp(k)$ is an existentially quantified variable for $0 \leq k < |\wp|$.

Combined Game

The parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ is a combination of the delayed automaton $\mathfrak{D}_{\mathcal{I}}^{\wp\psi}$ and the formula arena $\mathcal{A}_{\mathcal{I}}^{\wp\flat}$ (see Definition 5.7). This is very simple because our symbolic representations of both structures uses the same counter $\overline{d}$. The *edge relation* $E_{\mathfrak{G}}$ of the game is implemented symbolically as:

$$E_{\mathfrak{G}}(\overline{v},\overline{d},\overline{e},\overline{t},\overline{v'},\overline{d'},\overline{e'},\overline{t'}) = E(\overline{v},\overline{d},\overline{e},\overline{v'},\overline{d'},\overline{d'}) \wedge \mathcal{R}_D(\overline{t},\overline{d},\overline{v},\overline{t'},\overline{d'})$$

Again, the sets of *all vertices* $V_{\mathfrak{G}}$, *existential player vertices* $V_{\mathfrak{G}0}$, and *universal player vertices* $V_{\mathfrak{G}1}$ can be calculated symbolically as follows:

$$V_{\mathfrak{G}}(\overline{v},\overline{d},\overline{e},\overline{t}) = \mathrm{lfp}_{\Theta}\left[\left(\bigvee_{g\in G} g(\overline{v}) \wedge 0(\overline{e}) \wedge \mathcal{I}_D(\overline{d},\overline{t})\right) \vee \exists \overline{v'}\overline{d'}\overline{e'}\overline{t'}.\, (\Theta' \wedge E(\overline{v'},\overline{d'},\overline{e'},\overline{t'},\overline{v},\overline{d},\overline{e},\overline{t}))\right]$$

$$V_{\mathfrak{G}0}(\overline{v},\overline{d},\overline{e},\overline{t}) = V_{\mathfrak{G}}(\overline{v},\overline{d},\overline{e},\overline{t}) \wedge \bigvee_{k=0}^{|\wp|-1} k(\overline{d}) \wedge \mathsf{existential}(k)$$

$$V_{\mathfrak{G}1}(\overline{v},\overline{d},\overline{e},\overline{t}) = V_{\mathfrak{G}}(\overline{v},\overline{d},\overline{e},\overline{t}) \wedge \bigvee_{k=0}^{|\wp|-1} k(\overline{d}) \wedge \neg\mathsf{existential}(k)$$

The *colouring function*[9] $\mathcal{C}_{\mathfrak{G}}$ is represented as a family of $|\mathcal{C}|$ Boolean formulas:

$$\mathcal{C}_{\mathfrak{G}i}(\overline{v},\overline{d},\overline{e},\overline{t}) = V_{\mathfrak{G}}(\overline{v},\overline{d},\overline{e},\overline{t}) \wedge \mathcal{C}_i(\overline{t})$$

where $\mathcal{C}$ is the family of Boolean formulas representing the colouring function of the parity automaton $\mathfrak{P}_{\mathcal{I}}^{\psi}$ and $0 \leq i < |\mathcal{C}|$ a colour.

Once we have a symbolic representation of the parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$, we solve it using the existing algorithm in Figure 2.9. The symbolic implementation of the algorithm is quite straightforward because it uses only basic set operations (Subsection 2.3.2 explains how set operations can be represented symbolically) and attractors (see Definition 2.59). We shall therefore only discuss how *attractors* and *attractor strategies* can be calculated symbolically. Recall that an attractor $\mathsf{attr}_{\sigma}(\mathcal{A},X) \subseteq V$ of a set of nodes $X \subseteq V$ for a player $\sigma \in \{0,1\}$ in an arena $\mathcal{A} = (V_0,V_1,E)$ is defined as:

$$\mathsf{attr}_{\sigma}(\mathcal{A},X) \triangleq \mathrm{lfp}_Q\left[X \cup \{v \in V_{\sigma} \mid vE \cap Q \neq \emptyset\} \cup \{v \in V_{\overline{\sigma}} \mid vE \subseteq Q\}\right]$$

Intuitively, $\mathsf{attr}_{\sigma}(\mathcal{A},X)$ is the set of all vertices of $\mathcal{A}$ from which player σ can force a visit to X. If the vertices of $\mathcal{A}$ are represented using a Boolean vector[10] $\overline{u}$ (and its next version $\overline{u'}$), the attractor can be calculated symbolically as:

$$\mathsf{attr}_{\sigma}(\mathcal{A},X)(\overline{u}) = \mathrm{lfp}_{\Theta}\left[X(\overline{u}) \vee \left(V_{\sigma}(\overline{u}) \wedge \exists\overline{u'}.\,\Theta' \wedge E(\overline{u},\overline{u'})\right) \vee \left(V_{\overline{\sigma}}(\overline{u}) \wedge \neg\exists\overline{u'}.\,\neg\Theta' \wedge E(\overline{u},\overline{u'})\right)\right] \tag{5.2}$$

We are also interested in the winning strategies of $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ for the purposes of strategy synthesis (see Subsection 5.2.3). In parity games, these are constructed from attractor strategies. Informally, a (memoryless) attractor strategy for $\mathsf{attr}_{\sigma}(\mathcal{A},X)$ assigns to states $\mathsf{attr}_{\sigma}(\mathcal{A},X) \cap V_{\sigma}$ moves "towards" the target

[9]More formally, $\mathcal{C}_{\mathfrak{G}i}$ is a symbolic representation of the set of all vertices of $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ with colour $0 \leq i < |\mathcal{C}_{\mathfrak{G}}|$, i.e. $\mathcal{C}_{\mathfrak{G}i} = \{v \in V_{\mathfrak{G}} \mid c_{\mathfrak{G}}(v) = i\}$.

[10]In the case of the combined game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$, we have $\overline{u} = \overline{v}\overline{d}\overline{e}\overline{t}$.

set X. We can represent a strategy as a set of pairs (v, v') such that $v \in \mathsf{attr}_\sigma(\mathcal{A}, X) \cap V_\sigma$ and $v' \in vE$. Moreover, we can represent both the attractor and the strategy together as a disjunction of the attractor strategy (which contains all vertices $\mathsf{attr}_\sigma(\mathcal{A}, X) \cap V_\sigma$) and the vertices of the other player in the attractor $\mathsf{attr}_\sigma(\mathcal{A}, X) \cap V_{\overline{\sigma}}$. This can be calculated as:

$$\begin{aligned}\mathsf{attr}_\sigma(\mathcal{A}, X)(\overline{u}, \overline{u'}) = \mathrm{lfp}_\Theta \big[& X(\overline{u}, \overline{u'}) \vee \big(V_\sigma(\overline{u}) \wedge \neg \left(\exists \overline{u'}.\, \Theta\right) \wedge (\exists \overline{u}.\, \Theta') \wedge E(\overline{u}, \overline{u'})\big) \vee \\ & \big(V_{\overline{\sigma}}(\overline{u}) \wedge \neg \exists \overline{u'}.\, \neg\, (\exists \overline{u}.\, \Theta') \wedge E(\overline{u}, \overline{u'})\big)\big]\end{aligned}$$

Note that X can be a combination of vertices $v \in V$ and strategy mappings $(v, v') \in V \times V$ as well. This fixpoint is very similar to the one in Equation 5.2. The main difference is that Θ is now a Boolean formula over both $\overline{v}$ and $\overline{v'}$. Hence, we have to existentially quantify it *before* combining it with the edge relation (unlike in Equation 5.2), i.e. $(\exists \overline{u}.\, \Theta') \wedge E(\overline{u}, \overline{u'})$ instead of $\exists \overline{u'}.\, \Theta' \wedge E(\overline{u}, \overline{u'})$. More importantly, there is one extra term, namely $\neg \left(\exists \overline{u'}.\, \Theta\right)$, in the existential player case. It ensures that we add each state to the attractor only once. This is in line with the definition of the attractor strategy (see Definition 2.59). If we did not add this restriction, we could end up in the following situation: Let $\mathcal{A} = (V_0, V_1, E)$ be an arena with 3 vertices $V = \{a, b, c\}$ such that all of them belong to the existential player, i.e. $V_0 = V$ and $V_1 = \emptyset$. Furthermore, assume that the edge relation is $E = \{(b, a), (c, a), (b, c), (c, b)\}$. Consider the calculation of $\mathsf{attr}_0(\mathcal{A}, \{a\})$. Firstly, the pairs (b, a) and (c, a) are added to the initial set $\{a\}$. If we do not require that each vertex is added at most once, (b, c) and (c, b) will be added to the attractor in the second iteration. But then we would have $(b, c), (c, b) \in \mathsf{attr}_0(\mathcal{A}, \{a\})$ and $b, c \notin \{a\}$, i.e. a strategy which cycles between b and c forever and never reaches the target set $\{a\}$.

Model Checking and Strategy Synthesis

The symbolic implementation of the algorithm for solving parity games (see Figure 2.9) will return Boolean formulas representing both the winning regions and strategies $(W_{\mathfrak{G}0}, W_{\mathfrak{G}1})$ of the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$. The formula $\|\wp\flat\psi\|_{\mathcal{I}}(\overline{v})$ encoding the set of all global states of $\mathcal{I}$ at which the SL[1G] principal formula $\wp\flat\psi$ holds is finally calculated as (see Theorem 5.1):

$$\|\wp\flat\psi\|_{\mathcal{I}}(\overline{v}) = \exists \overline{d}\, \overline{e}\, \overline{t}\, \overline{v'}\, \overline{d'}\, \overline{e'}\, \overline{t'}.\, W_{\mathfrak{G}0}(\overline{v}, \overline{d}, \overline{e}, \overline{t}, \overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}) \wedge \underbrace{\mathcal{I}_D(\overline{d}, \overline{t}) \wedge 0(\overline{e})}_{\mathsf{start}}$$

The game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat\psi}$ together with the winning regions and strategies $(W_{\mathfrak{G}0}, W_{\mathfrak{G}1})$ encodes the synthesised strategies for all variables in the quantification prefix $\wp$. Assume that we want to determine the next action $f_k(\pi)$ that the strategy f_k for variable $\wp(k)$ with $0 \leq k < |\wp|$ should assign to the track $\pi \in Trk$ in $\mathcal{I}$. Given the next actions $f_0(\pi), \ldots, f_{k-1}(\pi)$ of strategies for variables $\wp(0), \ldots, \wp(k-1)$, we proceed as follows (see Definition 5.9):

1. We start in the state $s_0^0(\overline{v}, \overline{d}, \overline{e}, \overline{t}) = \mathsf{start}(\pi(0))(\overline{v}, \overline{d}, \overline{e}, \overline{t}) = \pi(0)(\overline{v}) \wedge \mathcal{I}_D(\overline{d}, \overline{t}) \wedge 0(\overline{e})$. We basically set the initial state t_I of the underlying parity automaton $\mathfrak{P}_{\mathcal{I}}^{\psi}$ of the game $(t_0 = t_I)$.

2. For $i = 1$ to $|\pi| - 1$: We set $s_i^0(\overline{v}, \overline{d}, \overline{e}, \overline{t}) = \exists \overline{v'}\, \overline{d'}\, \overline{e'}\, \overline{t'}.\, E_{\mathfrak{G}}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}, \overline{v}, \overline{d}, \overline{e}, \overline{t}) \wedge \pi(i-1)(\overline{v}) \wedge 0(\overline{d}) \wedge \left[\exists \overline{d'}\, \overline{e'}.\, s_{i-1}^0(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'})\right]$. Essentially, we are traversing the automaton $\mathfrak{P}_{\mathcal{I}}^{\psi}$ $(t_i = \delta(t_{i-1}, \pi(i-1)))$.

3. For $j = 0$ to $k-1$: We set $s_{|\pi|-1}^{j+1}(\overline{v}, \overline{d}, \overline{e}, \overline{t}) = \exists \overline{v'}\, \overline{d'}\, \overline{e'}\, \overline{t'}.\, E_{\mathfrak{G}}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}, \overline{v}, \overline{d}, \overline{e}, \overline{t}) \wedge s_{|\pi|-1}^{j}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}) \wedge f_j(\pi)(\overline{e^j})$. We accumulate the decision $[f_0(\pi), \ldots, f_{k-1}(\pi)]$.

4. $s_{|\pi|-1}^{k}$ is the implied current state of the combined parity game $\mathfrak{G}_{\mathcal{I}}^{\wp\flat}$ (given π and $f_0(\pi), \ldots f_{k-1}(\pi)$). Hence, we can use the winning regions and strategies $(W_{\mathfrak{G}0}, W_{\mathfrak{G}1})$ to determine the next action $f_k(\pi)$: We select an action $a \in \bigcap_{i \in \mathsf{sharing}(\flat\top, \wp(k))} P_i(l_{i\mathrm{E}}(\mathsf{last}(\pi)))$ such that the expression $\exists \overline{v}\, \overline{d}\, \overline{e}\, \overline{t}\, \overline{v'}\, \overline{d'}\, \overline{e'}\, \overline{t'}.\, E_{\mathfrak{G}}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}, \overline{v}, \overline{d}, \overline{e}, \overline{t}) \wedge \big(W_{\mathfrak{G}0}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}, \overline{v}, \overline{d}, \overline{e}, \overline{t}) \vee W_{\mathfrak{G}1}(\overline{v'}, \overline{d'}, \overline{e'}, \overline{t'}, \overline{v}, \overline{d}, \overline{e}, \overline{t})\big) \wedge a(\overline{e'^k})$ is not false. If there is no such action, we select any possible action a.

This concludes our symbolic implementation of SL[1G] model checking and strategy synthesis.

Complexity

The time complexity of our symbolic implementation is the same as that of the model checking algorithm discussed in Subsection 5.2.2, namely polynomial in the size of the model $|\mathcal{I}|$ and doubly exponential in the size of the formula $|\varphi|$.

Theorem 5.4. Let $\mathcal{I}$ be a an interpreted system and φ an SL[1G] sentence. The worst-case time complexity of the symbolic implementation is:

$$|\mathcal{I}|^{2^{O(|\varphi|)}}$$

Proof (Sketch). We only need to prove the statement for an arbitrary *basic principal sentence* $\varphi = \wp\flat\psi$ (see the proof of Theorem 5.2). The symbolic implementation of our algorithm uses:

- $O(\log_2 |G|)$ Boolean variables to represent the *global state*;
- $O(\sum_{i \in Agt} \log_2 |Act_i|)$ Boolean variables to represent *joint actions*;
- $O(|\psi|)$ Boolean variables to represent the states of the *non-deterministic (generalised) Büchi automaton*;
- $2^{O(|\psi|)}$ Boolean variables to represent the *deterministic parity automaton*;
- $O(\log_2 |\wp|)$ Boolean variables to represent the *delayed automaton and arena counter*;
- $O(\sum_{i \in Agt} \log_2 |Act_i|)$ Boolean variables to represent *decisions.*

The total number of Boolean variables needed is thus $O(\log_2 |G| + \sum_{i \in Agt} \log_2 |Act_i| + |\wp|) + 2^{O(|\psi|)} = O(\log_2 |\mathcal{I}|) + 2^{O(|\varphi|)}$. Since BDD operations take polynomial time with respect to the size of the relevant BDDs in the worst case [49], the worst-case time complexity of each BDD operation is $O(|\mathcal{I}|^k)2^{2^{O(|\varphi|)}}$ for some fixed constant $k > 0$. As we have stated in the proof of Theorem 5.2, the model checking algorithm has at most $|\mathcal{I}|^{2^{O(|\varphi|)}}$ steps. By multiplying these two expressions, we get the desired worst-case time complexity $|\mathcal{I}|^{2^{O(|\varphi|)}}$. □

5.2.5 Separate Determinisation

The SL[1G] model checking algorithm complexity proof (see Theorem 5.2) and the experimental results (see Section 6.4) indicate that one major bottleneck of the algorithm is parity game solving. In this section, we propose an optimisation technique, which addresses this problem by reducing the size of the game. Our experimental results demonstrate that it significantly reduces model checking time as well as memory usage (see Table 6.4b).

We motivate the optimisation by means of a simple example. Consider the following SL[1G] basic principal sentence (for some interpreted system $\mathcal{I}$ with agents $Agt \triangleq \{a, b\}$):

$$\beta \triangleq [\![x]\!]\langle\!\langle y\rangle\!\rangle(a,x)(b,y)[(p \to \mathsf{G}\,\mathsf{F}\,p) \wedge (q \to \mathsf{G}\,\mathsf{F}\,q) \wedge (r \to \mathsf{G}\,\mathsf{F}\,r)]$$

The underlying LTL formula of β is $\psi_\beta = (p \to \mathsf{G}\,\mathsf{F}\,p) \wedge (q \to \mathsf{G}\,\mathsf{F}\,q) \wedge (r \to \mathsf{G}\,\mathsf{F}\,r)$. The non-deterministic Büchi automaton $\mathfrak{B}^{\psi_\beta}_{\mathcal{I}}$ for ψ_β obtained on line 3 of our SL[1G] model checking algorithm in Figure 5.3 has 508 reachable states which are not dead ends. Therefore, 509 Boolean variables will be required to encode a single pair (S_i, m_i) in a state of the deterministic parity automaton $\mathfrak{P}^{\psi_\beta}_{\mathcal{I}}$ on line 4 of our SL[1G] model checking algorithm (the determinisation procedure we use is described in Subsection 2.4.4). As we shall see, at least three such pairs will be required to encode a state of $\mathfrak{P}^{\psi_\beta}_{\mathcal{I}}$. Hence, at least 1527 Boolean variables will be required to encode a state of the deterministic parity automaton. To put this into perspective, $\mathfrak{P}^{\psi_\beta}_{\mathcal{I}}$ will have at least $2^{1527} \approx 4.7 \times 10^{459}$ states. Comparing this with Table 2.2, we see that our symbolic implementation using BDDs will *not* be able to check β. This is a very negative result given that β is a relatively concise specification. It renders our SL[1G] model checking algorithm presented in Subsection 5.2.1 completely infeasible for verification and synthesis of large multi-agent systems with multiple constraints.

We propose an optimisation technique, which we refer to as *separate determinisation*, which allows us to tackle the problem above. Specifications of systems are usually given as conjunctions of LTL formulas [77]. Hence, we are interested in the verifications of SL[1G] principal sentences of the form:

$$\wp\flat\,[\psi_0 \wedge \psi_1 \wedge \cdots \wedge \psi_{n-1}]$$

We proceed as before and recursively replace all direct principal subformulas of the formula above with atoms (see Subsection 5.2.1). Hence we can assume $\psi_0, \ldots, \psi_{n-1}$ are LTL formulas without any loss of generality. Our aim is to construct a deterministic automaton $\mathfrak{S}^{\psi}_{\mathcal{I}}$ which accepts precisely those infinite paths $\pi \in Pth$ along which the LTL formula $\psi \triangleq \psi_0 \wedge \cdots \wedge \psi_{n-1}$ holds. Take an arbitrary infinite path $\pi \in Pth$. We have the following chain of equivalences:

$$\begin{aligned} \mathcal{I}, \pi \models_{\mathrm{LTL}} \psi \quad & \text{iff} \quad \forall i \in \{0, \ldots, n-1\} \,.\, \mathcal{I}, \pi \models_{\mathrm{LTL}} \psi_i \\ & \text{iff} \quad \forall i \in \{0, \ldots, n-1\} \,.\, \pi \in \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}}) \\ & \text{iff} \quad \pi \in \bigcap_{i=0}^{n-1} \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}}) \end{aligned}$$

Therefore, ψ holds along π iff it is accepted by the deterministic parity automaton $\mathfrak{P}^{\psi_i}_{\mathcal{I}} = (T_i, G, t_{Ii}, \delta_i, c_i)$ for each conjunct ψ_i with $0 \le i < n$. Consider the following *generalised parity automaton* (see Definition 2.39) obtained as a product of $\mathfrak{P}^{\psi_0}_{\mathcal{I}}, \ldots, \mathfrak{P}^{\psi_{n-1}}_{\mathcal{I}}$:

$$\mathfrak{S}^{\psi}_{\mathcal{I}} \triangleq \prod_{i=0}^{n-1} \mathfrak{P}^{\psi_i}_{\mathcal{I}} = \left(\prod_{i=0}^{n-1} T_i, G, \left(t_{I0}, \ldots, t_{I(n-1)}\right), \delta, \{c_1, \ldots, c_{n-1}\} \right)$$

where $\delta((t_0, \ldots, t_{n-1}), g) \triangleq (\delta_0(t_0, g), \ldots, \delta_{n-1}(t_{n-1}, g))$ for all $(t_0, \ldots, t_{n-1}) \in \prod_{i=0}^{n-1} T_i$ and $g \in G$. We claim that $\mathfrak{S}^{\psi}_{\mathcal{I}}$ accepts exactly those infinite paths in $\mathcal{I}$ which satisfy ψ.

Lemma 5.3. Let $\mathcal{I}$ be an interpreted system, $\psi_0, \ldots, \psi_{n-1}$ arbitrary LTL formulas such that $\psi \triangleq \bigwedge_{i=0}^{n-1} \psi_i$, and $\pi \in Pth$ a path in $\mathcal{I}$. Furthermore, let $\mathfrak{P}^{\psi_i}_{\mathcal{I}}$ be a deterministic parity automaton equivalent to ψ_i for each $0 \le i < n$. Then the following holds for the deterministic generalised parity automaton $\mathfrak{S}^{\psi}_{\mathcal{I}} \triangleq \prod_{i=0}^{n-1} \mathfrak{P}^{\psi}_{\mathcal{I}}$: $\pi \in \mathsf{Lang}(\mathfrak{S}^{\psi}_{\mathcal{I}})$ iff $\mathcal{I}, \pi \models_{\mathrm{LTL}} \psi$.

Proof. We can prove this statement directly using equivalences: By construction of $\mathfrak{S}^{\psi}_{\mathcal{I}}$, $\pi \in \mathsf{Lang}(\mathfrak{S}^{\psi}_{\mathcal{I}})$ iff $\pi \in \bigcap_{i=0}^{n-1} \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}})$. By definition of set intersection, $\pi \in \bigcap_{i=0}^{n-1} \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}})$ iff $\pi \in \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}})$ for all $0 \le i < n$. By our initial assumption, for all $0 \le i < n$, we have $\pi \in \mathsf{Lang}(\mathfrak{P}^{\psi_i}_{\mathcal{I}})$ iff $\mathcal{I}, \pi \models_{\mathrm{LTL}} \psi_i$. By LTL semantics (Definition 2.8), $\mathcal{I}, \pi \models_{\mathrm{LTL}} \psi$ iff $\mathcal{I}, \pi \models_{\mathrm{LTL}} \psi_i$ for all $0 \le i < n$. Our claim follows. □

Consider again the LTL formula $\psi_\beta = (p \to \mathsf{G}\,\mathsf{F}\,p) \wedge (q \to \mathsf{G}\,\mathsf{F}\,q) \wedge (r \to \mathsf{G}\,\mathsf{F}\,r)$. Each of the non-deterministic Büchi automata $\mathfrak{B}^{\psi_0}_{\mathcal{I}}$, $\mathfrak{B}^{\psi_1}_{\mathcal{I}}$, and $\mathfrak{B}^{\psi_2}_{\mathcal{I}}$ for the subformulas $\psi_0 = p \to \mathsf{G}\,\mathsf{F}\,p$, $\psi_1 = q \to \mathsf{G}\,\mathsf{F}\,q$, and $\psi_2 = r \to \mathsf{G}\,\mathsf{F}\,r$ has 13 reachable states which are not dead ends. The resulting deterministic parity automata $\mathfrak{P}^{\psi_0}_{\mathcal{I}}$, $\mathfrak{P}^{\psi_1}_{\mathcal{I}}$, and $\mathfrak{P}^{\psi_2}_{\mathcal{I}}$ each have 42 variables[11]. Therefore, the deterministic generalised parity automaton $\mathfrak{S}^{\psi_\beta}_{\mathcal{I}}$ equivalent to ψ_β has 126 Boolean variables and $2^{126} \approx 8.5 \times 10^{37}$ states. While this is still a very large number, it is much more manageable than the size of the deterministic parity automaton $\mathfrak{P}^{\psi_\beta}_{\mathcal{I}}$, which would have at least $2^{1527} \approx 4.7 \times 10^{459}$ states.

Once we have constructed the deterministic generalised parity automaton $\mathfrak{S}^{\psi}_{\mathcal{I}}$, the optimised SL[1G] model checking algorithm proceeds in a very similar manner to the original one in Figure 5.3: We construct the formula arena $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ and the delayed automaton $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$. Note that $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ is unchanged but $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ is now a generalised parity automaton. Finally, we construct the combined *generalised parity game* $\mathfrak{S}^{\wp\flat\psi}_{\mathcal{I}}$ and solve it using the existing algorithm in Figure 2.10. The complete optimised algorithm is shown in Figure 5.4.

[11]Their states require at least three pairs (S_i, m_i) (see Subsection 2.4.4). Hence, they each have $3 \times (13 + 1) = 42$ variables. This is also the reason why we know that the original parity automaton $\mathfrak{P}^{\psi_\beta}_{\mathcal{I}}$ equivalent to ψ_β would require at least three pairs (S_i, m_i) to encode a state.

```
function CheckSl[1g]Optimised($\wp\flat\psi$)
    $\psi_0 \wedge \cdots \wedge \psi_{n-1} := \psi$                                  ▷ Split conjuncts.
    for $i := 0, \ldots, n-1$ do
        $\mathfrak{A}^{\psi_i}_{\mathcal{I}}$ := GenBüchi($\psi_i$)
        $\mathfrak{B}^{\psi_i}_{\mathcal{I}}$ := Büchi($\mathfrak{A}^{\psi_i}_{\mathcal{I}}$)
        $\mathfrak{P}^{\psi_i}_{\mathcal{I}}$ := Parity($\mathfrak{B}^{\psi_i}_{\mathcal{I}}$)
    end for
    $\mathfrak{G}^{\psi}_{\mathcal{I}}$ := GenParity($\mathfrak{P}^{\psi_0}_{\mathcal{I}}, \ldots, \mathfrak{P}^{\psi_{n-1}}_{\mathcal{I}}$)      ▷ Calculate automaton product.
    $\mathfrak{D}^{\wp\psi}_{\mathcal{I}}$ := Delayed($\mathfrak{G}^{\psi}_{\mathcal{I}}, \wp$)      ▷ Convert to a delayed generalised parity automaton.
    $\mathcal{A}^{\wp\flat}_{\mathcal{I}}$ := Arena($\mathcal{I}, \wp, \flat$)
    $\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$ := Combine($\mathcal{A}^{\wp\flat}_{\mathcal{I}}, \mathfrak{D}^{\wp\psi}_{\mathcal{I}}$)      ▷ Combine into a generalised parity game.
    $(W_0, W_1)$ := SolveGenParity($\mathfrak{G}^{\wp\flat\psi}_{\mathcal{I}}$)      ▷ Solve generalised parity game (Figure 2.10).
    return $\{g \in G \mid \mathsf{start}(g) \in W_0\}$
end function
```

Figure 5.4: The optimised model checking algorithm for Sl[1g] which uses *separate determinisation*. The steps which differ from the original algorithm (see Figure 5.3) have comments.

This optimised algorithm does not improve the general $|\mathcal{I}|^{2^{O(|\varphi|)}}$ time complexity of Sl[1g] model checking (see Subsection 5.2.2) because it has no effect when ψ has only one conjunct, in which case the generalised parity game is a parity game with one colouring function, i.e. $\mathfrak{G}^{\psi}_{\mathcal{I}} = \mathfrak{P}^{\psi}_{\mathcal{I}}$. Nevertheless, the following more precise worst-case time complexity can be derived[12]:

$$|\varphi| + \sum_{\varphi' \in \mathsf{sub}(\varphi)} |\mathcal{I}|^{D(\varphi')} \binom{D(\varphi')}{D_0(\varphi'), \ldots, D_{N(\varphi')-1}(\varphi')}$$

where $N(\varphi')$ is the number of conjuncts in $\varphi' = \wp\flat\left[\psi_0 \wedge \cdots \wedge \psi_{N(\varphi')-1}\right]$, $D_i(\varphi') \triangleq 2^{O(\mathsf{temp}(\psi_i))}$ for $0 \leq i < N(\varphi')$, $D \triangleq \sum_{i=0}^{N(\varphi')-1} D_i(\varphi')$, $\binom{n}{k_0, \ldots, k_{n-1}} \triangleq \frac{n!}{\prod_{i=0}^{n-1} k_i!}$, and $\mathsf{temp}(\varphi')$ is the number of temporal operators in φ' (without strict principal subformulas). Note that this expression is equal to Equation 5.1 when each principal subsentence of φ has exactly one conjunct.

The symbolic implementation of the model checking algorithm (see Subsection 5.2.4) is modified appropriately. There is one extra issue, which is only relevant to Sl[1g] *strategy synthesis* (see Subsection 5.2.3). Recall that generalised parity games are not memoryless determined for the existential player in general (see Subsection 2.4.5), i.e. the winning strategy for the existential player requires finite memory, namely a counter for the current conjunct $0 \leq i < n$. We solve this problem by *(i)* augmenting the Boolean vectors representing the current and next game state with $\lceil \log_2 n \rceil$ Boolean variables for the counter and *(ii)* adding an auxiliary state after every temporal transition in which the existential player has the option to keep or increment the value of the counter:

$$\begin{tikzcd}
 & & (g', [\,], t', i) \\
\left(g, \left[a_0, \ldots, a_{|\wp|-1}\right], t, i\right) \arrow[r, "\text{temporal}", "\text{transition}"'] & (g', -, t', i) \arrow[ur, "\text{keep } i"] \arrow[dr, "\text{increment } i"'] & \\
 & & (g', [\,], t', (i+1) \bmod n)
\end{tikzcd}$$

5.3 Summary

In this chapter, we first gave a brief overview of Sl[1g] and then introduced a practical model checking algorithm for it. This is a very important result because there is currently no such algorithm. In fact, we are not even aware of any practical model checking algorithms for Atl*, which is strictly subsumed

[12]We use the fact that the time complexity of solving a generalised parity game with n vertices, m edges, and maximum colours $d_0, \ldots, d_{k-1}$ is $O(mn^d)\binom{\lceil d/2 \rceil}{\lceil d_0/2 \rceil, \ldots, \lceil d_{k-1}/2 \rceil}$ where $d = \sum_{i=0}^{k-1} d_i$ (see Proposition 2.11). Note that we ignore the coefficients because $D(\varphi')/2 = \sum_{i=0}^{k-1} D_i(\varphi')/2 = \sum_{i=0}^{k-1} 2^{O(\mathsf{temp}(\psi_i))-1} = \sum_{i=0}^{k-1} 2^{O(\mathsf{temp}(\psi_i))} = D(\varphi')$.

by SL[1G]. Therefore, we believe that we have put forward the first practical model checking algorithm for both SL[1G] and ATL*. Moreover, we showed that it has optimal worst-case time complexity, proved its correctness, and provided an efficient symbolic implementation. Since it supports general strategy synthesis, it can be used to automatically generate agents' behaviour from SL[1G] specifications.

The algorithm reduces the model checking of an SL[1G] basic principal sentence $\wp\flat\psi$ to the problem of solving a combined parity game, which is obtained as the product of a formula arena, which represents the interdependency of the quantified strategies in the prefix $\wp\flat$, and a delayed automaton, which accepts all paths through the arena which satisfy the underlying LTL formula ψ. We showed how the winning strategies of the combined parity game can be used for SL[1G] strategy synthesis. We also provided an optimisation technique called separate determinisation, which improves the performance of the algorithm on SL[1G] formulas of the form $\wp\flat\,[\psi_0 \wedge \cdots \wedge \psi_{n-1}]$ by determinising each conjunct separately and reducing the problem to solving a generalised parity game instead.

In Section 6.3, we will describe how we developed an extension of MCMAS which implements the SL[1G] model checking algorithm introduced in this chapter using BDDs.

Chapter 6

Implementation

A very important part of this project was the implementation of the novel model checking algorithms for SLK an SL[1G] presented in Sections 4.2 and 5.2 respectively. We developed both algorithms as extensions for the MCMAS model checker described in Subsection 2.5.2. To our best knowledge, there are *no other tools* which would support SL or any of its fragments such as SLK or SL[1G]. In our opinion, this makes our contribution even more valuable, as it demonstrates the feasibility of SL as a practical specification language rather than a mere theoretical construct. Our belief is further supported by the experimental results described in Section 6.4.

Extending an existing tool relieved us of the burden to design and implement all low-level procedures from scratch. Instead, it allowed us to focus on the important aspects of the new model checking algorithms and spend more time optimising their performance. On the other hand, we had to follow the existing design pattern of the tool and accept its shortcomings. We shall start by describing the existing functionality, usage, and architecture of MCMAS. We will then describe the new functionality of the SLK and SL[1G] extensions and explain which parts of the tool we had to modify in order to support each of the fragments. Finally, we present the experimental results we obtained on several scalable scenarios.

6.1 Existing Tool

MCMAS is a Model Checker for Multi-Agent Systems (MAS) developed at Imperial College London released under GNU Public Licence (GPL) [5, 8, 63]. A brief overview of its existing functionality has already been provided in Subsection 2.5.2, where it was compared with other existing verification tools. Here we shall describe it in more detail to give the reader a better idea of the original tool as well as the extensions which we developed as part of this project.

6.1.1 Functionality

MCMAS has the following features:

1. **Underlying framework.** The multi-agent systems in MCMAS are modelled as *interpreted systems* (see Definition 2.5), which naturally capture temporal, strategic, deontic (correct behaviour), and epistemic (knowledge) properties of a system. An important consequence of using this formalism is that all agents evolve synchronously. The semantics of the specifications languages supported by MCMAS (described below) are also defined with respect to interpreted systems (see Section 2.2).

   The reader should be familiar with the concept of interpreted systems by now as we have been using it heavily throughout Chapters 4 and 5. A very simple toy model and its formal representation as an interpreted system is provided in Section 3.3.

2. **Input format.** The description of interpreted systems is provided in the form of ISPL files. Such files use special syntax to describe the whole model including the agents, protocols, evolution functions, local variables, observable variables, initial states, and specifications to be checked. Please refer to [9] for the complete ISPL syntax. ISPL code for the toy model introduced in Section 3.3 is provided in Appendix A.

3. **Specification languages.** MCMAS supports Ctl (see Subsection 2.2.2) and Atl (see Subsection 2.2.4) augmented with epistemic operators expressing agents' knowledge (see Subsection 2.2.6) and deontic modalities for correct behaviour (not relevant for our project). Moreover, MCMAS admits arbitrary nesting of Ctl and Atl operators as well as the non-temporal modalities. In addition, basic fairness conditions[1] are supported. Ltl, Ctl*, Atl*, and Sl (including all syntactic fragments) are currently *not supported* by MCMAS.

4. **Model checking.** The main function of MCMAS is model checking of specifications against a model of a system. The tool automatically verifies and reports whether each formula holds in all initial states of the model or not. Thanks to its underlying efficient symbolic implementation (see Subsection 2.3.2), MCMAS can easily handle state spaces with more than 10^{20} possible states [63].

5. **Witness and counterexample executions.** In addition to reporting whether a formula is true or false in the given model, MCMAS can provide the user with sample executions justifying the truth value of the formula. If the formula is true, a *witness execution* is generated. Conversely, if the formula is false, a *counterexample execution* is generated. This is a very useful feature as it allows the user to identify and then fix the parts of the model in which the specification is not satisfied.

6. **Interactive simulation.** The tool provides an interactive simulation mode, in which the user selects an initial state and follows a possible evolution of the model by choosing the actions of all agents at each time step. The user can always also backtrack to the previous state. This feature does not provide automatic verification. Instead, it allows the user to manually check that the model is correct.

7. **Graphical interface.** Although MCMAS is a purely command-line tool with textual input and output, an Eclipse plugin supporting most of its features has been developed. It can currently perform model checking, interactive simulation, and witness/counterexample analysis. Its most prominent feature is visualisation of witness/counterexample executions.

The tool runs on most architectures including Linux, Windows, and Mac OS.

The latest publicly available release of MCMAS at the time of writing is version 1.2.1. We describe here version 1.1.0, which was available in October 2013 when our project forked off. The main difference between the two versions is the support for uniform Atl semantics in version 1.2.1 (uniformity is explained in Subsection 4.1.2).

6.1.2 Usage

We provide here a short introduction to the ISPL file format and the MCMAS command-line tool. Note that this subsection is in no way intended to be a complete manual for MCMAS. Our aim is to give the reader a basic idea of how the tool can be used for purposes relevant to our project. Please refer to the MCMAS manual [9] for a complete description of the ISPL Syntax and the command-line tool.

ISPL Syntax

An ISPL file contains both the multi-agent system and the properties to be verified. These are internally represented as an interpreted system (see Definition 2.5) and logic formulas (see Section 2.2) respectively. The file consists of the following sections [9]:

- `Semantics` *(optional)* – Statement which defines the evolution function semantics (see below). Two options are available: `MultiAssignment` (default) and `SingleAssignment`.

- `Agent` – Every agent of the multi-agent system is defined separately. Defining the environment is optional. Each specification has the following fields for an agent $i \in Agt$:

[1]A fairness condition is an arbitrary formula φ which rules out paths along which φ is not true infinitely often. This increases the expressiveness of Ctl and Atl with the ability to assume that certain unwanted behaviour will not occur (e.g. that no agent will hold a lock forever) [9]. Note that fairness constraints can be expressed naturally in Ltl as $[\mathsf{G}\,\mathsf{F}\,\varphi] \rightarrow \psi$ where ψ is the formula that we want to check.

   - `Vars` *(optional for environment)* – Internal variables of the agent. Three types of variables are supported in ISPL: Boolean (`x: boolean`), enumeration (`y: {a, b, c}`), and bounded integer (`z: 1 .. 4`). These variables represent the set of internal states L_i of the agent.
   - `Obsvars` *(environment only, optional)* – Variables observable by all agents. The types of variables are the same as in `Vars`. The set of local states $L_{j\mathrm{E}}$ of each agent $j \in Agt$ is induced by the the observable variables (of the environment) and the agent's internal variables.
   - `Lobsvars` *(other agents only, optional)* – Set of local variables of the environment that the agent can observe. This makes ISPL syntax more expressive than interpreted systems because different agents can observe different internal variables of the environment.
   - `RedStates` *(optional)* – Constraints on the correct behaviour of the agent for deontic modalities. This field is not relevant for our project.
   - `Actions` – Set of actions Act_i available to the agent.
   - `Protocol` – Protocol P_i of the agent, which maps local states $L_{i\mathrm{E}}$ to sets of actions 2^{Act_i}. It consists of lines of the form ***`condition`*** `:` `{`***`actions`***`}`, where ***`condition`*** is a condition on the local state and ***`actions`*** is a list of actions available to agent i if the condition holds. Note that the conditions on different lines need not be mutually exclusive. The keyword `Other` represents a condition which is true iff all other conditions fail.
   - `Evolution` – Evolution function t_i of the agent. Similarly to the protocol, it consists of lines of the form ***`next_state`*** `if` ***`condition`***, where ***`condition`*** is a condition on the local state of the agent and the joint action of all agents.

     If multi-assignment semantics is used (default), ***`next_state`*** refers to all local variables that change and exactly one condition line applies. If single-assignment semantics is used instead, ***`next_state`*** can refer to only one variable and, for each local variable, exactly one condition line applies. Please refer to the MCMAS manual [9] for the exact meaning of the two options.

- `Evaluation` – Atomic propositions AP and their assignment h over global states. For every atomic proposition, the evaluation contains an entry of the form ***`atom`*** `if` ***`condition`***, where ***`condition`*** is a condition on the current global state.
- `InitStates` – Boolean formula over all variables which defines the initial states I.
- `Groups` *(optional)* – List of groups of agents. These are used in some epistemic modalities (see Subsection 2.2.6) and ATL temporal operators (see Subsection 2.2.4).
- `Fairness` *(optional)* – List of fairness formulas. This field is not relevant to our project.
- `Formulae` – List of formulas to be verified. Table 6.1 shows how logic formulas are translated into ISPL syntax.

Command-Line Tool

The general usage of the MCMAS command-line tool is as follows:

```
$ ./mcmas [OPTIONS] FILE
```

where *`[OPTIONS]`* is an optional list of command-line options and *`FILE`* is the name of the ISPL file. For example, the file `examples/toy_model_existing.ispl` contains the toy model introduced in Section 3.3, a CTL specification $\mathsf{EF}\,p$, and an ATL specification $\langle\langle\{1,2\}\rangle\rangle\mathsf{G}\,\neg\,(p_1 \vee p_2)$ (see Subsections 2.2.2 and 2.2.4 for a description of CTL and ATL respectively). These formulas are expressed in the ISPL file as `EF p` and `<g>G !(p1 or p2)` respectively, where `g` is a group containing agents `Player1` and `Player2`. The complete ISPL file is provided in Appendix A. The following command checks the specifications and generates witness/counterexample executions where possible:

```
$ ./mcmas -c 1 examples/toy_model_existing.ispl
```

The `-c` flag for displaying witness/counterexample executions has a numeric parameter `1`–`3` which specifies the format of the executions. This command generates the following output:

Logic	Formula	ISPL syntax
propositional	p	*atom*
	$\neg\varphi$	!*formula*
	$\varphi_1 \wedge \varphi_2$	*formula1* and *formula2*
	$\varphi_1 \vee \varphi_2$	*formula1* or *formula2*
	$\varphi_1 \rightarrow \varphi_2$	*formula1* -> *formula2*
Ctl (Subsection 2.2.2)	$\mathsf{AX}\,\varphi$	AX *formula*
	$\mathsf{EX}\,\varphi$	EX *formula*
	$\mathsf{AF}\,\varphi$	AF *formula*
	$\mathsf{EF}\,\varphi$	EF *formula*
	$\mathsf{AG}\,\varphi$	AG *formula*
	$\mathsf{EG}\,\varphi$	EG *formula*
	$\mathsf{A}[\varphi_1\,\mathsf{U}\,\varphi_2]$	A(*formula1* U *formula2*)
	$\mathsf{E}[\varphi_1\,\mathsf{U}\,\varphi_2]$	E(*formula1* U *formula2*)
Atl (Subsection 2.2.4)	$\langle\!\langle A\rangle\!\rangle\mathsf{X}\,\varphi$	<*group*>X *formula*
	$\langle\!\langle A\rangle\!\rangle\mathsf{F}\,\varphi$	<*group*>F *formula*
	$\langle\!\langle A\rangle\!\rangle\mathsf{G}\,\varphi$	<*group*>G *formula*
	$\langle\!\langle A\rangle\!\rangle[\varphi_1\,\mathsf{U}\,\varphi_2]$	<*group*>(*formula1* U *formula2*)
epistemic (Subsection 2.2.6)	$\mathsf{K}_i\,\varphi$	K(*agent*, *formula*)
	$\mathsf{E}_A\,\varphi$	GK(*group*, *formula*)
	$\mathsf{D}_A\,\varphi$	DK(*group*, *formula*)
	$\mathsf{C}_A\,\varphi$	GCK(*group*, *formula*)
deontic (not relevant)	$\mathsf{O}_i\,\varphi$	O(*agent*, *formula*)

Table 6.1: Translation between logic formulas and ISPL syntax.

```
************************************************************************
                       MCMAS v1.1.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permitted by applicable law.

 Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
 Please send any feedback to <mcmas@imperial.ac.uk>
************************************************************************

Command line: ./mcmas -c 1 examples/toy_model_existing.ispl

examples/toy_model.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building BDD for initial states...
Building reachable state space...
Checking formulae...
Verifying properties...
  Formula number 1: (EF p1), is TRUE in the model
  The following is a witness for the formula:
   < 0 1 >
  States description:
------------- State: 0 -----------------
Agent Environment
  state = game
Agent Player1
```

```
30 Agent Player2
31 ----------------------------------------
32 ------------- State: 1 -----------------
33 Agent Environment
34   state = p1win
35 Agent Player1
36 Agent Player2
37 ----------------------------------------
38   Formula number 2: (<g>G (! (p1 || p2))), is TRUE in the model
39   The following is a witness for the formula:
40 A witness exists but could not be generated.
41 done, 2 formulae successfully read and checked
42 execution time = 0.007
43 number of reachable states = 3
44 BDD memory in use = 8968368
```

Lines 22 and 38 indicate that the both the CTL formula $\mathsf{EF}\, p_1$ and the ATL formula $\langle\langle\{1,2\}\rangle\rangle \mathsf{G}\, \neg\, (p_1 \vee p_2)$ are true in all initial states of the model. A witness execution with 2 states, namely g_{g} and g_1 (see Section 3.3), for the CTL formula is provided on lines 24–37. According to line 40, MCMAS could not generate a witness execution for the ATL formula.

Other flags supported by MCMAS include deadlock checking (`-k`), arithmetic overflow checking (`-a`), interactive simulation (`-s`), and verbosity level (`-v` with a numeric parameter `1`–`5`). The complete list of flags supported by MCMAS is provided on the tool's help screen:

```
$ ./mcmas -h
```

6.1.3 Architecture

MCMAS uses binary decision diagrams (see Subsection 2.3.1) to represent the models of systems and implements all algorithms using very efficient symbolic operations on this data structure (see Subsection 2.3.2). The tool is written in C++ (except for the Eclipse plugin, which is implemented in Java) and uses the CUDD BDD package [85]. Over the years, it has been developed collaboratively by members of the Verification of Autonomous Systems Group (VAS) in the Department of Computing at Imperial College London [13].

The aim of this subsection is to provide a high-level description of MCMAS internals and bridge the gap between the theoretical background presented in Chapter 2 and the existing MCMAS functionality described in Subsection 6.1.1. We will first give an overview of the model checking and witness/counterexample generation process and then describe the overall structure of the MCMAS source code. Finally, we will comment on the software design of MCMAS.

Model Checking Process

Assume that MCMAS is executed with the name of an ISPL file as a command-line argument (see Subsection 6.1.2 for a short introduction to MCMAS usage). The tool performs the following steps:

1. **Parser.** The ISPL file is scanned and parsed into an interpreted system $\mathcal{I}$ representation of the model and an abstract syntax tree for each formula $\varphi_0, \ldots, \varphi_{n-1}$. The scanner and parser were automatically generated using Flex [4] and Bison [2] respectively.

2. **Syntax check.** The program checks that the the model and the specifications are well-formed. This includes type checking and name resolution. The tool ensures that all expressions (e.g. an agent's protocol) refer only to existing variables observable by the corresponding agent. If the model or a formula violates any of the constraits, an error message is printed and the whole execution is aborted.

3. **Symbolic encoding.** Each local variable (of an agent) with k possible values is allocated $2 \times \lceil \log_2 k \rceil$ BDD variables to represent its current and next value. Once all local variables of all agents have been allocated BDD variables, the BDD vectors $\overline{v}$, $\overline{v'}$ for representing the current and next

global state are constructed as their concatenation. Similarly, the BDD vector $\overline{u}$ for representing joint actions is composed of the BDD variables for representing individual actions. Consequently, protocols, evolution functions, initial states, and epistemic accessibility relations are calculated. A detailed example of how all of this can be done is provided in Subsection 3.3.2.

4. **Reachability analysis.** The set of states G reachable from the set of initial states I of the interpreted system $\mathcal{I}$ is calculated. As explained at the end of Subsection 3.3.2, this is performed using a simple fixpoint calculation over the temporal transition relation.

   Checks for deadlocks and/or arithmetic overflow in the reachable states are carried out at this stage if the user requests them via command-line flags.

   Finally, if there is at least one fairness condition in the ISPL file, the set of reachable fair states, at which some path satisfying all fairness conditions starts, is calculated.

5. **Model checking.** For each formula φ_i with $0 \leq i < n$, MCMAS checks if all initial states I of the interpreted system $\mathcal{I}$ satisfy φ_i, i.e. $I \subseteq \|\varphi_i\|_{\mathcal{I}}$. It does this by introducing an atom $init \in AP$ which holds in the initial states, i.e. $h(init) = I$, and then checking that the formula $init \rightarrow \varphi_i$ is true in all reachable states, i.e. $\|init \rightarrow \varphi_i\|_{\mathcal{I}} = G$.

   The tool uses a recursive bottom-up model checking algorithm similar to the one for CTL in Definition 2.12.

6. **Witness/counterexample generation.** If a formula starting with an E path quantifier is true in the model (e.g. $\mathcal{I} \models \mathsf{EG}\, p$), a witness execution is presented. Conversely, if a formula starting with an A path quantifier does not hold in the model (e.g. $\mathcal{I} \not\models \mathsf{AF}\, p$), a counterexample execution is presented. Both witness and counterexample executions are provided for Boolean combinations of formulas. Please refer to [9] for more details about the combinations of CTL and ATL operators supported by witness/counterexample generation in MCMAS.

   Note that witness/counterexample generation is performed only when explicitly requested by the user via a command-line flag.

Please refer to the MCMAS source code available at [8] for low-level details of the individual steps.

Source Code Structure

The structure of the MCMAS source code directory is as follows (not all files are listed):

- `main.cc` – Main file of the whole tool. It uses functions and classes defined in the other files to implement the model checking process we discussed earlier. It is responsible for the majority of output and defines global variables used by the other source files.
- `Makefile` – Project build file. This file describes how MCMAS is built from the source files. The whole tool can be compiled easily by executing the `make` command. Build rules for a wide range of platforms including Linux, Windows (using Cygwin), and Mac OS are provided. Both 32 and 64-bit architectures are supported.
- `cudd-2.5.0-exp/` – The CUDD package for manipulating BDDs.
- `doc/` – MCMAS documentation,
- `examples/` – Sample ISPL files.
- `include/` – Header files.
- `parser/` – Files relevant to ISPL file parsing (e.g. token and grammar files) and syntax checking.
- `utilities/` – All other source files.
  - `computereach.cc` – Functions for calculating the set of reachable states.
  - `modal_formula.cc` – Class representing modal formulas. This source file also implements the recursive model checking algorithm and provides the witness/counterexample generation functionality.

- `read_options.cc` – Command-line arguments handler.
- `simulation.cc` – Interactive simulation.
- `utilities.cc` – Model checking functions and custom BDD operations.

Software Design

Overall, MCMAS source code uses a mixture of the *object-oriented paradigm* and *procedural paradigm*:

- Object-oriented paradigm: classes, objects, inheritance, encapsulation
- Procedural paradigm: functions, global variables

As we have already pointed out in Section 1.2, the quality of the code is quite low from a software engineering point of view, despite being developed as open source. We have encountered the following issues during the development of the extensions:

1. **Complete lack of testing.** We believe that this is the *biggest problem* by far as there are no guarantees about the correctness of individual functions, classes, or the program as a whole. Furthermore, it makes any large-scale code refactoring virtually impossible because the programmer has no easy way of verifying that they did not break any existing functionality. It is almost paradoxical that a software verification tool is not verified in any structured way.

2. **Global variables.** The source code of MCMAS uses many global variables, which are generally believed to be a bad programming practice. Consequently, it is difficult to test individual functions and reason about their correctness because of side-effects.

3. **Inconsistent style.** Different parts of the codebase use different indent styles. In addition, some files contain lines with more than 100 characters. This makes the source code difficult to read and understand.

4. **Long functions.** Some of the functions are very long. For example, the body of the method for generating witness/counterexample executions has 2053 lines. It is very difficult to understand and modify such functions.

Ideally, the existing source code should be refactored and augmented with tests. Unfortunately, the current situation is a vicious cycle: Refactoring is difficult without having tests and adding tests is difficult without refactoring first. Therefore, we believe that the whole tool should be redesigned and rewritten from scratch in the future.

Despite these shortcomings, MCMAS is on of the leading tools in the area of symbolic model checking of multi-agent systems. More importantly, its existing source code provided us with solid foundations on which we could build the SLK and SL[1G] extensions. In the next two sections, we will describe how we extended MCMAS with support for SLK and SL[1G], which were discussed in Chapters 4 and 5 respectively.

6.2 Epistemic Strategy Logic Extension

Epistemic Strategy Logic (SLK) is a new fragment of SL, which we introduced in Chapter 4. As it is defined on imperfect recall semantics with incomplete information, it can combine various epistemic and game-theoretic concepts. Consequently, SLK can express complex specifications such as an agent's knowledge about Nash equilibria. In Section 4.2 we provided a model checking algorithm for SLK as well as an efficient symbolic implementation of it. We have developed an extension of MCMAS which puts the algorithm into practice and thereby adds support for SLK model checking to MCMAS. We will now discuss the new functionality of the extension, its usage, and the modified architecture.

SLK Formula	ISPL syntax
$\langle\langle x\rangle\rangle\varphi$	`<<variable>> formula`
$[\![x]\!]\varphi$	`[[variable]] formula`
$(i,x)\varphi$	`(agent, variable) formula`
$\mathsf{X}\,\varphi$	`X formula`
$\mathsf{F}\,\varphi$	`F formula`
$\mathsf{G}\,\varphi$	`G formula`
$\varphi_1 \,\mathsf{U}\, \varphi_2$	`formula1 U formula2`

Table 6.2: Translation between SLK formulas and ISPL syntax. Translation of the logics supported by the original version of MCMAS is shown in Table 6.1.

6.2.1 Functionality

The SLK extension of MCMAS, which we developed as part of this project, has the following new features (see Subsection 6.1.1 for the list of existing features):

1. **Specification languages** *(updated)***.** Our extension adds support for SLK specifications to MCMAS. Moreover, it can handle arbitrary nesting of CTL, ATL, and SLK operators.

2. **Model checking** *(updated)***.** We have incorporated the efficient symbolic implementation of our SLK model checking algorithm (see Subsection 4.2.4) in the extension.

   In order to improve its performance, we have developed an *experimental parallel implementation* of the SLK model checking algorithm which can be enabled using a command-line flag (see Subsection 6.2.2).

3. **Witness and counterexample executions** *(updated)***.** We have added support for SLK operators to the witness/counterexample execution generator.

4. **Witness and counterexample strategies** *(new)***.** We have implemented witness and counterexample strategy synthesis for SLK formulas as explained in Subsection 4.2.3. If an SLK formula $\langle\langle x_0\rangle\rangle \dots \langle\langle x_{m-1}\rangle\rangle\psi$ holds in the model, witness strategies for the variables $x_0, \dots, x_{m-1}$ are synthesised (when requested by the user). Conversely, if an SLK formula $[\![y_0]\!] \dots [\![y_{n-1}]\!]\phi$ does not hold in the model, counterexample strategies for the variables $y_0, \dots, y_{n-1}$ are synthesised.

To sum up, the extension adds support for SLK together with witness and counterexample strategy synthesis. This is a major enhancement of MCMAS functionality as it provides the possibility to automatically synthesise agents' behaviour which satisfies an SLK specification.

6.2.2 Usage

The SLK extension augments ISPL syntax with the new SLK operators (see Definition 4.1). The translation from SLK formulas to ISPL syntax is shown in Table 6.2. Apart from that, the usage of the SLK extension is the same as that of the original tool (see Subsection 6.1.2).

Consider the file `examples/toy_model_slk.ispl` containing the toy model introduced in Section 3.3 and the SLK specification $\langle\langle e\rangle\rangle(\mathrm{E}, e)\langle\langle x\rangle\rangle(1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$, which means that *"there exist strategies for all agents such that neither player will ever win"*. This formula is expressed in ISPL syntax as `<<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y) G !(p1 or p2)`. The complete ISPL file is provided in Appendix A. The following command checks the specification and generates a witness/counterexample execution with strategies (where possible):

```
$ ./mcmas -c 1 examples/toy_model_slk.ispl
```

The SLK extension generates the following output:

```
************************************************************************
                          MCMAS-SL v1.1.0

```

```
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 examples/toy_model_slk.ispl
12
13 examples/toy_model.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 Verifying imperfect recall properties...
22   Formula number 1: <<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y)
23                     G (! (p1 || p2)), is TRUE in the model
24   The following is a witness for the formula:
25    < 0 0 >
26   States description:
27 ------------- State: 0 -----------------
28 Agent Environment
29   state = game
30 Agent Player1
31 Agent Player2
32 ----------------------------------------
33   Strategies:
34 ------- Strategy e [Environment] -------
35 Agent Environment
36   Environment.state=game (0): idle
37   Environment.state=p1win (1): idle
38   Environment.state=p2win (2): idle
39 ----------------------------------------
40 --------- Strategy x [Player1] ---------
41 Agent Player1
42   Environment.state=game (0): rock
43   Environment.state=p1win (1): idle
44   Environment.state=p2win (2): idle
45 ----------------------------------------
46 --------- Strategy y [Player2] ---------
47 Agent Player2
48   Environment.state=game (0): rock
49   Environment.state=p1win (1): idle
50   Environment.state=p2win (2): idle
51 ----------------------------------------
52 done, 1 imperfect recall formulae successfully read and checked
53 execution time = 0.006
54 number of reachable states = 3
55 BDD memory in use = 8989648
```

Line 23 indicates that the SLK formula is true. A witness execution, namely a cycle in the state g_g, is provided on lines 25–32. More importantly, the following *witness strategies* for the SLK specification are provided on lines 34–51:

$$
\begin{array}{lll}
f_e(g_{\mathrm{g}}) \triangleq \mathrm{i} & f_x(g_{\mathrm{g}}) \triangleq \mathrm{r} & f_y(g_{\mathrm{g}}) \triangleq \mathrm{r} \\
f_e(g_1) \triangleq \mathrm{i} & f_x(g_1) \triangleq \mathrm{i} & f_y(g_1) \triangleq \mathrm{i} \\
f_e(g_2) \triangleq \mathrm{i} & f_x(g_2) \triangleq \mathrm{i} & f_y(g_2) \triangleq \mathrm{i}
\end{array}
$$

The extension adds a new command-line flag `-t`, which enables the experimental parallel implementation of the SLK model checking algorithm. It has a non-negative numeric parameter which defines the maximum depth of thread branching, e.g. `-t 3` allows at most $2^3 = 8$ threads. This functionality is disabled by default.

6.2.3 Architecture

In order to add SLK support to MCMAS, we had to modify its existing source code and add new files. We will first give a high-level overview of how we updated the steps of the original model checking process. Then we shall list the modified and new source files. Finally, we will discuss the experimental parallel implementation of the model checking algorithm.

Model Checking Process

The original MCMAS model checking process (see Subsection 6.1.3) has been modified in the SLK extension as follows:

1. **Parser.** The ISPL token and grammar files were augmented with SLK syntax (see Table 6.2).

2. **Syntax check.** The updated syntax checking procedure verifies that all SLK specifications are sentences (see Definition 4.1), i.e. that all agents are bound to a strategy when a temporal transition occurs.

3. **Symbolic encoding.** The symbolic encoding of the global states and joint actions is unchanged. Since the number of BDD variables needed to represent variable assignments (see Definition 4.7) depends on the SLK formula, they are allocated for each SLK specification separately in the recursive model checking algorithm (step 5). More precisely, BDD variables for representing the strategy assigned to a variable x are allocated when the corresponding quantifier $\langle\langle x \rangle\rangle$ or $[\![x]\!]$ is encountered.

4. **Reachability analysis.** This step is completely unchanged in the extension.

5. **Model checking.** We have augmented the existing recursive symbolic model checking algorithm of MCMAS with the symbolic implementation of our SLK model checking algorithm presented in Subsection 4.2.4.

6. **Witness/counterexample generation.** We added support for SLK to the existing procedure for generating witness/counterexample executions. More importantly, we implemented SLK witness/counterexample strategy synthesis, which we discussed in Subsection 4.2.3.

Please refer to the attached project source code for low-level details of the individual steps.

Source Code Structure

We modified and augmented the original MCMAS codebase (see Subsection 6.1.3) while developing the SLK extension. The following files were added or updated (not all files are listed):

- `main.cc` – We made only minor changes to the main file (e.g. added a function for printing witness strategies).

- `Makefile` – We added build rules for the new files.

- `examples/` – We added several sample ISPL files which demonstrate the new functionality of the extension.

- `include/` – We included prototype declarations for new classes in the existing header files. In addition, we added separate header files for benchmarking and the Slk model checking algorithm functions.
- `parser/` – We modified the ISPL token and grammar files.
- `utilities/` – All new features were implemented in this folder:
  - `benchmark.cc` *(new)* – Toolkit for analysing the performance of both extensions (Slk and Sl[1g]). We developed it specifically for this project in order to find bottlenecks in the algorithms.
  - `modal_formula.cc` – We extended the syntax checking procedure, the recursive model checking algorithm, and the function for generating witnesses/counterexamples.
  - `read_options.cc` – We added the `-t` command-line flag, which enables experimental parallelisation of Slk model checking.
  - `sl_imperfect_recall.cc` *(new)* – This file contains the core Slk model checking procedures, which are invoked by the recursive model checking algorithm in `modal_formula.cc`. The core functionality includes calculating the shared state space of a quantifier, allocating BDD variables for strategies, and Slk temporal operators. This file also provides the experimental parallel implementation of the model checking algorithm.
  - `strategy.cc` *(new)* – Class representing Slk strategies.

Parallel Implementation

As we have explained in Subsection 4.2.4, our symbolic implementation of the Slk model checking algorithm represents a strategy $f \in \mathit{UStr}_A$ for a set of agents $A \subseteq \mathit{Agt}$ as a mapping from shared local states $S \in G/\sim^{\mathsf{C}}_A$ to actions. In other words, for every shared local state $S \in G/\sim^{\mathsf{C}}_A$, we store the corresponding action using a separate set of BDD variables. Therefore, we must explicitly enumerate the shared local state space. The algorithm which performs this computation for a given set of agents $A \subseteq \mathit{Agt}$ is shown in Figure 6.1.

Our experimental results indicate that the enumeration of local states (function LocalStates in Figure 6.1) is a *major bottleneck* of the algorithm. In order to improve the performance of our algorithm, we created an experimental *parallel implementation* of the procedure which uses multiple threads. It spawns a separate thread for one of the two recursive function calls on line 17 of the LocalStates function in Figure 6.1. Each time such a branching occurs, both threads are left with roughly one half of the original state space. In order not to create too many threads, branching only occurs in the t top-most recursive calls, where t is the maximum depth of thread branching (set via the `-t` command-line flag). More precisely, a new thread is spawned only when $n < t$, where n is the fourth argument of LocalStates and represents the current recursion depth. The impact of this optimisation on the performance of the model checking algorithm is shown in Table 6.3.

While the results indicate that the optimisation does reduce model checking time, the approach appears not to be very scalable. The problem is that the CUDD BDD package has *no support for concurrency*. In order to avoid race conditions, each thread has to have a separate BDD manager[2]. After all the local states are enumerated by the threads, the corresponding BDDs (one per local state) must be *transferred* to a single BDD manager. Since each manager might be using a completely different variable ordering (see Subsection 2.3.1), this transfer soon becomes the bottleneck when multiple threads are used. The only solution to this problem, apart from rewriting the CUDD library, is to use a different BDD package which supports concurrency. Given the fact that MCMAS already uses CUDD extensively and the limited time for our project, we decided not pursue this direction any further. Instead, we devoted all of our remaining time to the development of a model checking algorithm for Sl[1g], which we then implemented as another extension for MCMAS (see the next section).

[2]A *BDD manager* stores a hash table for BDD nodes, which ensures the canonicity of the BDDs, and other auxiliary data structures. Every BDD is handled by a BDD manager. More importantly, the arguments of BDD operations must have the same manager.

```
function SharedStates($A$, $\overline{v}$, $G$)                ▷ Enumerate shared local states of agents $A$.
    for $k := 0$ to $|A| - 1$ do
        $\hat{L}_{A(k)} :=$ LocalStates($A(k)$, $\overline{v}$, $G(\overline{v})$, 0)
    end for
    $S := \hat{L}_{A(0)}$
    for $k := 1$ to $|A| - 1$ do
        $S :=$ Merge($S$, $\hat{L}_{A(k)}$)
    end for
    return $S$
end function

function LocalStates($i$, $\overline{v}$, $\lambda$, $n$)                ▷ Enumerate local states of agent $i$.
    if $\lambda = \bot$ then
        return $\emptyset$                                    ▷ $\lambda$ is not reachable.
    else if $n = |\overline{v_{i\mathrm{E}}}|$ then
        return $\{\lambda\}$
    else
        return LocalStates($i$, $\overline{v}$, $\lambda \wedge \neg\overline{v_{i\mathrm{E}}}(n)$, $n+1$) $\cup$ LocalStates($i$, $\overline{v}$, $\lambda \wedge \overline{v_{i\mathrm{E}}}(n)$, $n+1$)
    end if
end function

function Merge($S_1$, $S_2$)                                ▷ Merge state spaces $S_1$ and $S_2$.
    for $j := 0$ to $|S_2| - 1$ do
        $S := \{s \in S_1 \mid s(\overline{v}) \wedge S_2(j)(\overline{v}) \neq \bot\}$     ▷ Find states $S \subseteq S_1$ compatible with $S_2(j)$
        $S_1 := S_1 \setminus S$                            ▷ Remove the compatible states $S$.
        $S_1 := S_1 \cup \left\{S_2(j)(\overline{v}) \vee \bigvee_{s\in S} s(\overline{v})\right\}$     ▷ Merge $S_2(j)$ and the compatible states $S$.
    end for
    return $S_1$
end function
```

Figure 6.1: Functions for enumerating the set of *shared local states* $S = G/{\sim}_A^{\mathsf{C}}$ for a set of agents $A \subseteq Agt$. $\overline{v}$ is a Boolean vector for representing the current global state and G is the set of global states.

6.3 One-Goal Strategy Logic Extension

One-goal Strategy Logic (SL[1G]) is another fragment of SL, which we discussed in Chapter 5. Unlike SLK, it is defined with respect to perfect recall semantics and complete information, i.e. agents have both perfect memory of the past and complete knowledge of the system. In Section 5.2 we provided a model checking algorithm for SL[1G] as well as an efficient symbolic implementation of it. Furthermore, we have developed an extension of MCMAS which puts the algorithm into practice and thereby adds support for SL[1G] model checking and strategy synthesis to MCMAS. We will now discuss the new functionality of the extension, its usage, and the modified architecture.

6.3.1 Functionality

The SL[1G] extension of MCMAS, which we developed as part of this project, has the following new features (see Subsection 6.1.1 for the list of existing features):

1. **Specification languages** *(updated)*. Our extension adds support for SL[1G] specifications to MCMAS. The new SL[1G] syntax is separate from the other logics (CTL, ATL, and SLK) because the fragment does not support epistemic modalities (due to complete information) and uses a very different model checking algorithm based on ω-automata (see Section 5.2).

2. **Model checking** *(updated)*. We have incorporated the efficient symbolic implementation of our SL[1G] model checking algorithm (see Subsection 5.2.4) in the extension. Hence, it now supports the verification of SL[1G] specifications.

   We implemented both the original and the *optimised version* of our algorithm (see Subsection 5.2.5).

3. **Strategy synthesis** *(new)*. We implemented full SL[1G] strategy synthesis, which we discussed in Subsection 5.2.3 and explained how it can implemented symbolically in Subsection 5.2.4. Unlike the SLK extension (see Subsection 6.2.1), the SL[1G] extension supports strategy synthesis for *arbitrary* SL[1G] formulas. However, unlike SLK witness and counterexample strategies, the synthesised strategies may be inter-dependent, i.e. the next action of one strategy in the current state might depend on the next actions of other strategies (see Subsection 5.2.3).

4. **Witness and counterexample strategies** *(new)*. Although witness and counterexample strategies for formulas of the form $\langle\langle x_0\rangle\rangle \ldots \langle\langle x_{m-1}\rangle\rangle\psi$ and $[[y_0]] \ldots [[y_{n-1}]]\phi$ are technically subsumed by full strategy synthesis (previous point), the extension provides the option to present them differently. Rather then generating a full solution with inter-dependent strategies for all variables (see Figure 5.2), a separate strategy is constructed for each variable in the existential or universal quantification prefix. Both representations are demonstrated in Subsection 6.3.2.

To sum up, the extension adds support for SL[1G] model checking and strategy synthesis. Similarly to the SLK extension presented in Section 6.2, this is a major enhancement of MCMAS functionality as it provides the possibility to automatically synthesise agents' behaviour which satisfies an SL[1G] specification.

As we have pointed out in Section 3.2, SL[1G] strictly subsumes ATL*. Therefore, all logics in the ATL* hierarchy which were previously not supported by MCMAS, notably LTL, CTL*, and ATL* (see Subsections 2.2.1, 2.2.3, and 2.2.4), are now indirectly[3] supported thanks to our extension. Moreover, the extension has optimal time complexity[4] for ATL* model checking and strategy synthesis (see Table 2.1 and Theorem 5.4). This is a major achievement because there are *no existing tools* that would support ATL* specifications (as far as we know).

[3]The formulas must be rewritten into equivalent SL[1G] formulas in order to be checked by MCMAS, e.g. an LTL formula ψ has to be transformed to the equivalent SL[1G] formula $[[x_0]] \ldots [[x_{|Agt|-1}]] (Agt(0), x_0) \ldots (Agt(|Agt|-1), x_{|Agt|-1})\psi$. This is only a matter of adding extra ISPL syntax which would automatically perform the translation.

[4]This is not the case for LTL and CTL*, which both have PSPACE-COMPLETE model checking complexity with respect to the size of the formula (see Table 2.1).

6.3.2 Usage

The SL[1G] extension augments ISPL syntax with the new SL[1G] operators (see Definition 5.2). The translation from SL[1G] formulas to ISPL syntax is the same as for SLK formulas (see Table 6.2). Unlike SLK formulas, SL[1G] formulas cannot contain operators of the other modal logics supported by MCMAS. In order to avoid ambiguity (SLK and SL[1G] use the same operators), every SL[1G] formula must be preceded by a `#PR` (perfect recall) tag in an ISPL file:

```
1  ...
2  Formulae
3    -- SLK formula
4    <<e>> (Environment, e) <<x>> (Player, x) <<y>> (Player2, y) G !(p1 or p2)
5    -- SL[1G] formula
6    #PR <<e>> (Environment, e) <<x>> (Player, x) <<y>> (Player2, y) G !(p1 or p2)
7  end Formulae
8  ...
```

The `Formulae` section contains the formula $\langle\langle e\rangle\rangle(\mathrm{E}, e)\langle\langle x\rangle\rangle(1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$ twice: Line 4 represents an SLK formula (imperfect recall with incomplete information) whereas line 6 represents an SL[1G] formula (perfect recall with complete information). Apart from the `#PR` tag, the usage of the SL[1G] extension is the same as that of the original tool and the SLK extension (see Subsections 6.1.2 and 6.2.2).

Witness/Counterexample Strategy Synthesis

We first demonstrate the ability of the SL[1G] extension to synthesise witness/counterexample strategies and its limitations. Consider the file `examples/toy_model_s1g.ispl` containing the toy model introduced in Section 3.3 and the SL[1G] specification $\langle\langle e\rangle\rangle(\mathrm{E}, e)\langle\langle x\rangle\rangle(1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$, which means that *"there exist strategies for all agents such that neither player will ever win"*. This formula is expressed in ISPL syntax as `#PR <<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y) G !(p1 or p2)`. The complete ISPL file is provided in Appendix A. The following command checks the specification and generates witness/counterexample strategies:

```
$ ./mcmas -c 1 examples/toy_model_sl1g.ispl
```

The SL[1G] extension generates the following output (additional line breaks were inserted so that the output would fit on a page):

```
************************************************************************
                       MCMAS-SL v1.1.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permitted by applicable law.

 Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
 Please send any feedback to <mcmas@imperial.ac.uk>
************************************************************************

Command line: ./mcmas -c 1 examples/toy_model_sl1g.ispl

examples/toy_model_sl1g.ispl has been parsed successfully.
Global syntax checking...
Done
  Global reachable states:
    g0: Environment.state=game (initial)
    g1: Environment.state=p1win
    g2: Environment.state=p2win
  Perfect recall formula 1: <<e>> (Environment, e) <<x>> (Player1, x) <<y>>
                            (Player2, y) G (! (p1 || p2)), is TRUE in the model
```

```
Witness 1/1 (<<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y)
             G (! (p1 || p2))):
  Strategy <<e>>:
    start:
      g0->s0 (initial)
      g1->s1
      g2->s2
    s0 (winning):
      action: idle
      g0->s3
      g1->s4
      g2->s5
    s1:
      action: <no winning action>
      g1->s6
    s2:
      action: <no winning action>
      g2->s7
    s3 (winning):
      action: idle
      g0->s3
      g1->s4
      g2->s5
    s4:
      action: <no winning action>
      g1->s6
    s5:
      action: <no winning action>
      g2->s7
    s6:
      action: <no winning action>
      g1->s6
    s7:
      action: <no winning action>
      g2->s7
  Strategy <<x>>:
    start:
      g0->s0 (initial)
      g1->s1
      g2->s2
    s0 (winning):
      action: paper
      g0->s3
      g1->s4
      g2->s5
    s1:
      action: <no winning action>
      g1->s6
    s2:
      action: <no winning action>
      g2->s7
    s3 (winning):
      action: paper
      g0->s3
      g1->s4
      g2->s5
```

```
78       s4:
79         action: <no winning action>
80         g1->s6
81       s5:
82         action: <no winning action>
83         g2->s7
84       s6:
85         action: <no winning action>
86         g1->s6
87       s7:
88         action: <no winning action>
89         g2->s7
90     Strategy <<y>>:
91       start:
92         g0->s0 (initial)
93         g1->s1
94         g2->s2
95       s0 (winning):
96         action: paper
97         g0->s3
98         g1->s4
99         g2->s5
100      s1:
101        action: <no winning action>
102        g1->s6
103      s2:
104        action: <no winning action>
105        g2->s7
106      s3 (winning):
107        action: paper
108        g0->s3
109        g1->s4
110        g2->s5
111      s4:
112        action: <no winning action>
113        g1->s6
114      s5:
115        action: <no winning action>
116        g2->s7
117      s6:
118        action: <no winning action>
119        g1->s6
120      s7:
121        action: <no winning action>
122        g2->s7
123  done, 1 perfect recall formulae successfully read and checked
124  execution time = 0.015
125  number of reachable states = 3
126  BDD memory in use = 9163152
```

Line 21 indicates that the SL[1G] formula is true. Witness strategies for the variables e, x, and y are provided on lines 24–122. Unlike SLK witness strategies (see Subsection 6.2.2), which are memoryless, SL[1G] witness strategies require *finite memory.* Thus, they are presented to the user in the form of automata where each automaton state corresponds to one memory state. Since the SL[1G] formula is true in the initial state g_{g}, the underlying LTL formula $\mathsf{G}\,\neg\,(p_1 \vee p_2)$ will be true along the path if all agents follow the corresponding witness strategies.

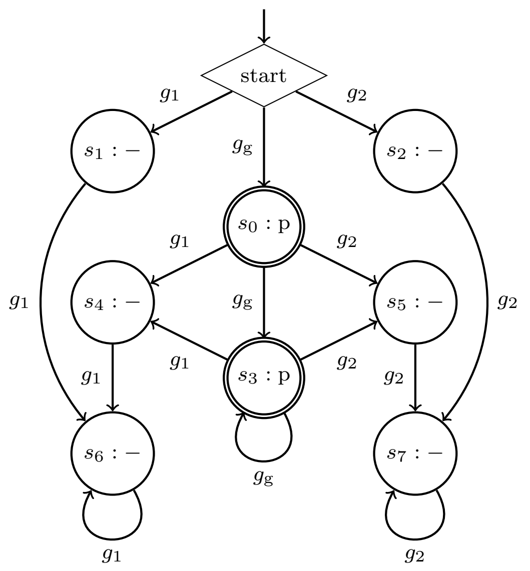


Figure 6.2: Finite memory witness strategy for variable x in the SL[1G] formula $\langle\langle e\rangle\rangle(\mathrm{E}, e)\langle\langle x\rangle\rangle(1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$ synthesised by the SL[1G] extension for the toy model. Winning memory states have double borders and contain the actions that should be performed next, e.g. paper should be played in the memory state s_3. Memory updates are represented by transitions labelled with the new global states.

We shall now explain how the witness strategies synthesised by the extension work. The automaton representation of the strategy for variable x is shown in Figure 6.2. The initial memory state of the automaton depends on the initial global state. This dependency is represented by the start node. For example, if the initial global state is g_g, the initial memory state is s_0. If the SL[1G] formula can be enforced from a memory state, it is referred to as a winning memory state and is assigned the next action to perform, e.g. the next action in s_3 is paper. The memory state is updated upon every temporal transition with the new global state. For example, if the current memory state is s_3 and a temporal transition to the global state g_1 occurs, the new memory state is s_4. Note that neither the next action nor the memory update depends directly on the current global state.

Witness strategies can only be synthesised for existential quantification prefixes $\langle\langle x_0\rangle\rangle \ldots \langle\langle x_{m-1}\rangle\rangle$ in states where the formula holds. Conversely, counterexample strategies can only be synthesised for universal quantification prefixes $[\![y_0]\!] \ldots [\![y_{n-1}]\!]$ in states where the formula does not hold. Consider the much more interesting SL[1G] sentence $\gamma \triangleq [\![e]\!](\mathrm{E}, e)[\![x]\!](1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$ used throughout Chapter 5, which means roughly: *"Whichever action player 1 performs, there exists an action for player 2 such that neither player will ever win"*. Counterexample strategies can be synthesised for the universal quantification prefix $[\![e]\!][\![x]\!]$ of γ in states g_1 and g_2, where it does not hold. However, we are more interested in synthesising a strategy for player 2 (variable y). Unfortunately, this cannot be done using witness strategy synthesis because the strategy depends on the strategies for the environment and player 1 (variables e and x). General strategy synthesis, which we describe next, has to be used in this case.

General Strategy Synthesis

We now demonstrate the more general ability of the SL[1G] extension to synthesise arbitrary strategies. Consider the file `examples/toy_model_sl1g2.ispl` containing the toy model introduced in Section 3.3 and the SL[1G] specification $\gamma \triangleq [\![e]\!](\mathrm{E}, e)[\![x]\!](1, x)\langle\langle y\rangle\rangle(2, y)\mathsf{G}\,\neg\,(p_1 \vee p_2)$ discussed earlier. This formula is expressed in ISPL syntax as `#PR [[e]] (Environment, e) [[x]] (Player1, x) <<y>> (Player2, y) G !(p1 or p2)`. The complete ISPL file is provided in Appendix A. The following command checks the specification γ and synthesises strategies for all variables in it:

```
$ ./mcmas -solutions examples/toy_model_sl1g2.ispl
```

where `-solutions` is a new command-line flag introduced by the SL[1G] extension which enables general strategy synthesis. The following output is generated by the extension (additional line breaks were inserted so that the output would fit on a page):

```
************************************************************************
                       MCMAS-SL v1.1.0

 This software comes with ABSOLUTELY NO WARRANTY, to the extent
 permitted by applicable law.

 Please check http://vas.doc.ic.ac.uk/tools/mcmas/ for the latest release.
 Please send any feedback to <mcmas@imperial.ac.uk>
************************************************************************

Command line: ./mcmas -solutions examples/toy_model_sl1g2.ispl

examples/toy_model_sl1g2.ispl has been parsed successfully.
Global syntax checking...
Done
Encoding BDD parameters...
Building partial transition relation...
Building BDD for initial states...
Building reachable state space...
Checking formulae...
Verifying imperfect recall properties...
done, 0 imperfect recall formulae successfully read and checked
Verifying perfect recall properties...
  Global reachable states:
    g0: Environment.state=game (initial)
    g1: Environment.state=p1win
    g2: Environment.state=p2win
  Perfect recall formula 1: [[e]] (Environment, e) [[x]] (Player1, x) <<y>>
                            (Player2, y) G (! (p1 || p2)), is TRUE in the model
  Solution 1/1 ([[e]] (Environment, e) [[x]] (Player1, x) <<y>> (Player2, y)
                G (! (p1 || p2))):
    start:
      g0->s0 (initial)
      g1->s1
      g2->s2
    s0 (winning):
      temporal transition:
        g0->s3
        g1->s4
        g2->s5
      strategies:
        [[e]]=idle:
          [[x]]=paper:
            <<y>>=paper*: g0
            <<y>>=rock: g1
            <<y>>=scissors: g2
          [[x]]=rock:
            <<y>>=paper: g2
            <<y>>=rock*: g0
            <<y>>=scissors: g1
          [[x]]=scissors:
            <<y>>=paper: g1
          <<y>>=rock: g2
          <<y>>=scissors*: g0
    s1:
      temporal transition:
        g1->s6
      strategies:
        [[e]]=idle*:
          [[x]]=idle*:
            <<y>>=idle: g1
    s2:
      temporal transition:
        g2->s7
      strategies:
        [[e]]=idle*:
          [[x]]=idle*:
            <<y>>=idle: g2
    s3 (winning):
      temporal transition:
        g0->s3
        g1->s4
        g2->s5
      strategies:
        [[e]]=idle:
          [[x]]=paper:
            <<y>>=paper*: g0
            <<y>>=rock: g1
            <<y>>=scissors: g2
          [[x]]=rock:
            <<y>>=paper: g2
            <<y>>=rock*: g0
            <<y>>=scissors: g1
          [[x]]=scissors:
            <<y>>=paper: g1
            <<y>>=rock: g2
            <<y>>=scissors*: g0
    s4:
      temporal transition:
        g1->s6
      strategies:
        [[e]]=idle*:
          [[x]]=idle*:
            <<y>>=idle: g1
    s5:
      temporal transition:
        g2->s7
      strategies:
        [[e]]=idle*:
          [[x]]=idle*:
            <<y>>=idle: g2
    s6:
      temporal transition:
        g1->s6
      strategies:
        [[e]]=idle*:
          [[x]]=idle*:
            <<y>>=idle: g1
```

```
109     s7:
110       temporal transition:
111         g2->s7
112       strategies:
113         [[e]]=idle*:
114           [[x]]=idle*:
115             <<y>>=idle: g2
116 done, 1 perfect recall formulae successfully read and checked
117 execution time = 0.046
118 number of reachable states = 3
119 BDD memory in use = 9195888
```

Line 29 indicates that γ is true in the initial state g_g. The synthesised strategies are provided on lines 32–115 in the form of one large *automaton* which encodes both the memory and interdependencies between strategies. Consider the memory state s_0, which is winning, i.e. γ can be enforced by the existentially quantified strategies. It has two fields:

1. `temporal transition`. This field defines how the memory state is updated upon temporal transition. The memory update is the same as for witness SL[1G] strategies presented earlier.

   For example, if the current memory state is s_0 and the next global state is g_0, the next memory state is s_3.

2. `strategies` This field represents how the next action of each strategy depends on the actions of other strategies. If an existential strategy can enforce γ from its current memory state, its next action is marked with `*`. Conversely, if a universal strategy can falsify γ from its current memory state, its next action is also marked with `*`. The next global state for each combination of actions is also provided for convenience.

   For example, if the current memory state (of the strategy for variable y) is s_0 and the next actions of strategies for variables e and x are idle and scissors respectively, the next action of the strategy for variable y is scissors. The next global state will then be g_0, which in turn determines the next memory state s_3 (according to the `temporal transition` field).

Observe that the synthesised solution has a very similar structure to the combined parity game $\mathfrak{G}^{\gamma}_{\mathcal{I}}$ in Figure 5.2. This is a direct consequence of the SL[1G] strategy synthesis procedure discussed in Subsection 5.2.3.

Separate Determinisation

The separate determinisation optimisation technique for the SL[1G] model checking algorithm proposed in Subsection 5.2.5 is *enabled* by default. It can be disabled using the command-line flag `-sd 0`.

6.3.3 Architecture

In order to add SL[1G] support to MCMAS, we had to modify its existing source code and add new files. We will first give a high-level overview of how we updated the steps of the original model checking process and then list the modified and new source files.

Model Checking Process

The original MCMAS model checking process (see Subsection 6.1.3) has been modified in the SL[1G] extension as follows:

1. **Parser.** The ISPL token and grammar files were augmented with SL[1G] syntax (see Table 6.2).

2. **Syntax check.** Since SL[1G] formulas cannot be mixed with other logics and have a distinct syntax (thanks to the `#PR` tag), we implemented the syntax check as a separate procedure. It verifies that all SL[1G] specifications are sentences that satisfy the syntactic constraints (see Definition 5.2).

3. **Symbolic encoding.** The symbolic encoding of the global states and joint actions is unchanged. Since the number of BDD variables needed to represent the auxiliary model checking data structures (see Subsection 5.2.1) depends on the SL[1G] formula, they are allocated for each SL[1G] specification separately in the model checking algorithm (step 5).
4. **Reachability analysis.** This step is completely unchanged in the extension.
5. **Model checking.** We implemented the symbolic model checking algorithm for SL[1G] presented in Subsection 5.2.4. Similarly to the syntax checking procedure (step 2), the algorithm is separate from the one for the other logics (including SLK).
6. **Witness/counterexample generation.** The method for SL[1G] witness/counterexample strategy synthesis is again separate from the one for the other logics supported by MCMAS. It reuses the solution calculated by the procedure for general strategy synthesis (step 7).
7. **General strategy synthesis.** We implemented general strategy synthesis symbolically as explained in Subsection 5.2.4. Since this functionality is not provided by either the original tool or the SLK extension, it was developed as a completely new procedure.

Please refer to the attached project source code for low-level details of the individual steps.

Source Code Structure

We modified and augmented the original MCMAS codebase (see Subsection 6.1.3) while developing the SL[1G] extension. The following files were added or updated (not all files are listed):

- `main.cc` – We made only minor changes to the main file (e.g. added a function for printing reachable states).
- `Makefile` – We added build rules for the new files.
- `examples/` – We added several sample ISPL files which demonstrate the new functionality of the extension.
- `include/` – We included prototype declarations for new classes in the existing header files. In addition, we added separate header files for benchmarking and the SL[1G] model checking algorithm functions and classes.
- `parser/` – We modified the ISPL token and grammar files.
- `utilities/` – All new features were implemented in this folder:
  - `arena.cc`, `automaton.cc`, `game.cc`, `generalized_automaton.cc`, `intermediate_automaton.cc`, `nondeterministic_automaton.cc`, `parity_automaton.cc` *(new)* – Classes representing auxiliary data structures used by the SL[1G] model checking algorithm (see Subsection 5.2.1). A UML diagram for all these classes is shown in Figure 6.3.
  - `benchmark.cc` *(new)* – Toolkit for analysing the performance of both extensions (SLK and SL[1G]). We developed it specifically for this project in order to find bottlenecks in the algorithms.
  - `modal_formula.cc` – We implemented the (separate) syntax checking procedure for SL[1G] formulas.
  - `quantifier.cc` *(new)* – Class representing a strategy quantifier (e.g. $\langle\langle x\rangle\rangle$).
  - `read_options.cc` – We added the `-solutions` and `-sd` command-line flags, which enable general strategy synthesis and toggle separate determinisation respectively.
  - `sl_perfect_recall.cc` *(new)* – This is the main file of the SL[1G] extension. It contains the implementation of the recursive model checking algorithm with strategy synthesis (see Subsection 5.2.4).
  - `sl_solution.cc` *(new)* – Class representing the solution of an SL[1G] formula. It contains *(i)* the set of global states in which the formula holds and *(ii)* the synthesised strategies with interdependencies represented as one large automaton (see Subsection 6.3.2).

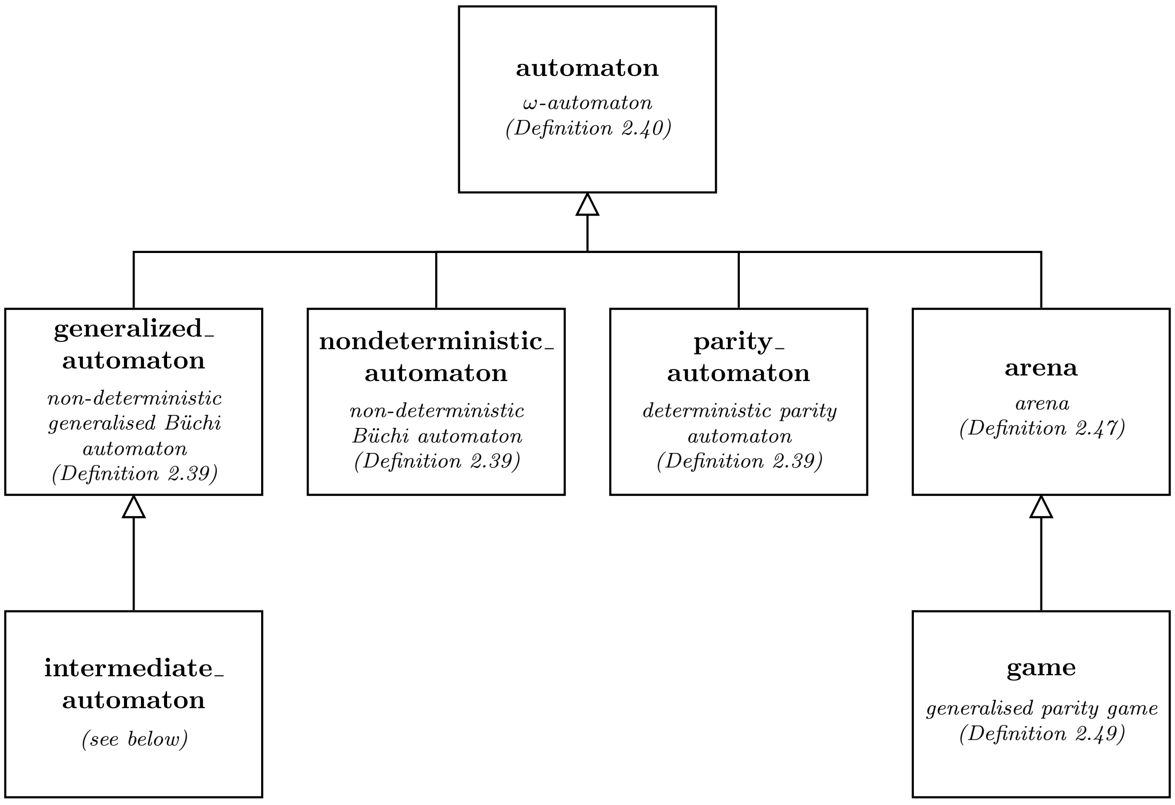


Figure 6.3: UML diagram for classes representing auxiliary data structures used by the SL[1G] model checking algorithm. The `intermediate_automaton` class represents a more general form of a non-deterministic generalised Büchi automaton, which we use for intermediate results in the standard translation (see Subsection 2.4.3).

6.4 Experimental Results

We present here the experimental results obtained using the Slk and Sl[1g] extensions discussed in Sections 6.2 and 6.3 respectively on several scalable real-life scenarios. We measure the amount of time and memory used by both extensions and compare it with the performance of the original tool on Ctl and Atl. We also demonstrate the differences in expressiveness and strategy synthesis between the two fragments.

The following scalable real-life scenarios are discussed in this section:

1. **Dining Cryptographers.** A simple security protocol which ensures anonymity of participants. We use this example to demonstrate the ability of Slk to express agents' knowledge and compare the performance of the tool on equivalent Ctlk, Atlk, and Slk formulas. We also use it to evaluate the impact of the experimental parallel implementation of the Slk model checking algorithm, which we discussed at the end of Subsection 6.2.3.

2. **Cake cutting.** A problem where a cake of a certain size needs to be divided fairly among a group of agents. It demonstrates not only the ability of Slk to express Nash equilibria, but also the fact that our Slk extension can synthesise protocols which achieve them.

3. **Scheduler.** A process scheduler which ensures mutual exclusion of a shared resource while preventing starvation. We use this example to compare the performance of Ctl, Atl, Slk, and Sl[1g] model checking. We also evaluate the impact of the separate determinisation optimisation technique for the Sl[1g] model checking algorithm proposed in Subsection 5.2.5.

4. **Nim.** A traditional game in which players take turns to remove objects from heaps until all heaps are empty. We use this example to compare the performance of Atl, Slk, and Sl[1g] model checking on a fixed-size specification.

The experiments were run on an Intel® Core^{TM} i7-3770 CPU 3.40GHz lab machine with 16GB RAM running Linux kernel version `3.8.0-35-generic`. ISPL files and results for all scenarios can be found in the `examples/benchmarks` folder in the attached project archive. In addition, the source code of automatic ISPL code generators for some of the scenarios can be found in the `examples/generators` folder.

6.4.1 Dining Cryptographers

The dining cryptographers protocol is a hypothetical protocol introduced in [29] where the anonymity of participants is preserved due to lack of knowledge:

> *"Three cryptographers are sitting down to dinner at their favourite 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 NSA (U.S. National Security Agency). The three cryptographers respect each other's right to make an anonymous payment, but they wonder if NSA is paying. They resolve their uncertainty fairly by carrying out the following protocol:*
>
> *Each cryptographer flips an unbiased coin behind his menu, between him and the cryptographer on his right, so that only the two of them can see the outcome. Each cryptographer then states aloud whether the two coins he can see—the one he flipped and the one his left-hand neighbour flipped—fell on the same side or on different sides. If one of the cryptographers is the payer, he states the opposite of what he sees. An odd number of differences uttered at the table indicates that a cryptographer is paying; an even number indicates that NSA is paying (assuming that the dinner was paid for only once). Yet if a cryptographer is paying, neither of the other two learns anything from the utterances about which cryptographer it is."*

The protocol described above works for an arbitrary number of cryptographers $n \geq 3$. We model it as an interpreted system (see Definition 2.5) with agents $Agt \triangleq \{\mathrm{E}, \mathrm{c}_1, \ldots, \mathrm{c}_n\}$. We model each cryptographer as an agent whose actions represent the possible announcements ("same" and "different"). The coins are internal variables of the Environment which we make selectively observable to the agents via their `Lobsvars` fields (see Subsection 6.1.2):

```
Agent DinCrypt1
  Lobsvars = {coin1, coin2};
  ...
end Agent
```

The required property of the protocol—if a cryptographer did not pay and an odd number of differences is uttered at the table, then *(i)* he knows that another cryptographer paid for the dinner but *(ii)* he does not know which one it was—is expressed in CTLK, ATLK, and SLK as:

$$\varphi_{\mathrm{CTLK}} \triangleq \mathsf{AG}\,\psi \qquad \varphi_{\mathrm{ATLK}} \triangleq \langle\!\langle \emptyset \rangle\!\rangle \mathsf{G}\,\psi \qquad \varphi_{\mathrm{SLK}} \triangleq [\![x_e]\!][\![x_1]\!]\cdots[\![x_n]\!](\mathrm{E},x_e)(\mathrm{c}_1,x_1)\cdots(\mathrm{c}_n,x_n)\mathsf{G}\,\psi \tag{6.1}$$

where[5]:

$$\psi \triangleq (\mathit{odd} \wedge \neg \mathit{paid}_1) \rightarrow \underbrace{\left[\mathsf{K}_{\mathrm{c}_1} \bigvee_{i=2}^{n} \mathit{paid}_i\right]}_{\substack{\text{cryptographer } \mathrm{c}_1\\ \textit{knows} \text{ that another}\\ \text{cryptographer paid}}} \wedge \underbrace{\left[\bigwedge_{i=2}^{n} \neg \mathsf{K}_{\mathrm{c}_1} \mathit{paid}_i\right]}_{\substack{\text{cryptographer } \mathrm{c}_1\\ \textit{does not know} \text{ which}\\ \text{cryptographer paid}}}$$

The model checking results for the formulas above with $3 \leq n \leq 18$ cryptographers in Table 6.3 and Figure 6.4 indicate that reasonably large state spaces can be verified by the SLK extension. We can observe that it has similar running time and peak memory usage to CTLK and ATLK for $n < 10$ cryptographers. However, its performance drops considerably faster for $n \geq 10$ cryptographers because of the large number of shared local states that need to be enumerated (see the discussion at the end of Subsection 6.2.3). Consequently, the SLK model checking procedure runs out of physical memory (16GB) for $n > 18$ cryptographers. In contrast, CTLK and ATLK require no assignments and their performance[6] is dominated by the calculation of the reachable state space.

Table 6.3 and Figure 6.4 also show the impact of the *experimental parallel implementation* of the SLK extension discussed at the end of Subsection 6.2.3. More precisely, we evaluated the performance of the SLK algorithm with 1, 2, and 4 threads (thread branching 0, 1, and 2 set using the `-t` command-line flag). We can observe that the 2-thread version is approximately 16% faster than the sequential one for $n \geq 10$ cryptographers[7]. Although the 4-thread version is slightly faster than the sequential one for $n \geq 11$ cryptographers as well, it does not outperform the 2-thread version for $n < 18$ cryptographers. This result confirms our prior belief that the optimisation approach is not very scalable (see Section 6.2.3). The execution of the 4-thread version on the problem with $n = 18$ cryptographers was prematurely terminated[8], possibly because it used too much memory (3.8GB) and/or CPU (166%).

Note that the protocol specification cannot be expressed in SL[1G] because it does not support epistemic modalities expressing agents' knowledge due to complete information semantics (see Section 5.1).

6.4.2 Cake Cutting

The cake-cutting problem is a well-known mathematical puzzle [41]:

> *"Two persons own a cake which they want to split into two parts to be allotted between them. The cake may be made of different ingredients with different values to each person, i.e. each person has his own measure for evaluating any given part of the cake. Is there any procedure or protocol which will enable the two persons to cut the cake into two pieces such that each person will get at least $\frac{1}{2}$ of the cake by his own measure?"*

We consider a generalisation of the problem [26, 27] where $n \in \mathbb{N}$ agents want to split a cake of size $d \in \mathbb{N}$. The problem is modelled as a multi-player game with n agents and an environment, which has to provide a protocol to slice the cake in a fair way. The game has *hybrid* turns: at even rounds the n

[5]By symmetry of the problem, we can consider the first cryptographer without loss of generality.

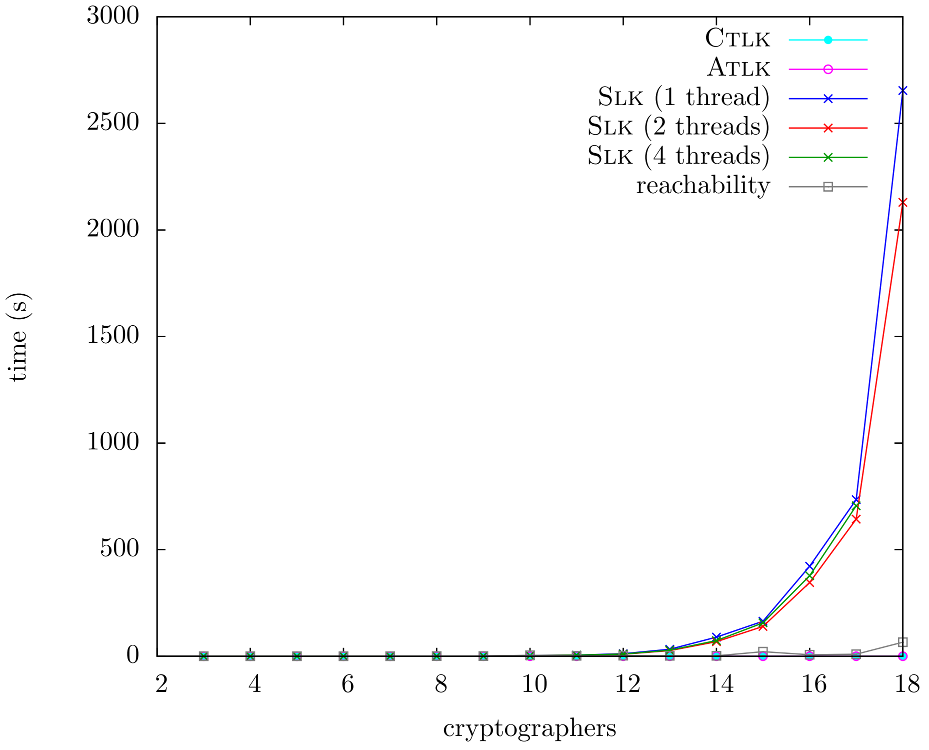
[6]Surprisingly, the ATLK model checking algorithm is always faster than the one for CTLK (despite ATLK being strictly more expressive than CTLK). However, this difference is negligible compared with the running time of the reachable state space calculation.

[7]Incidentally, this is the same threshold from which SLK performance becomes considerably lower than that of CTLK and ATLK.

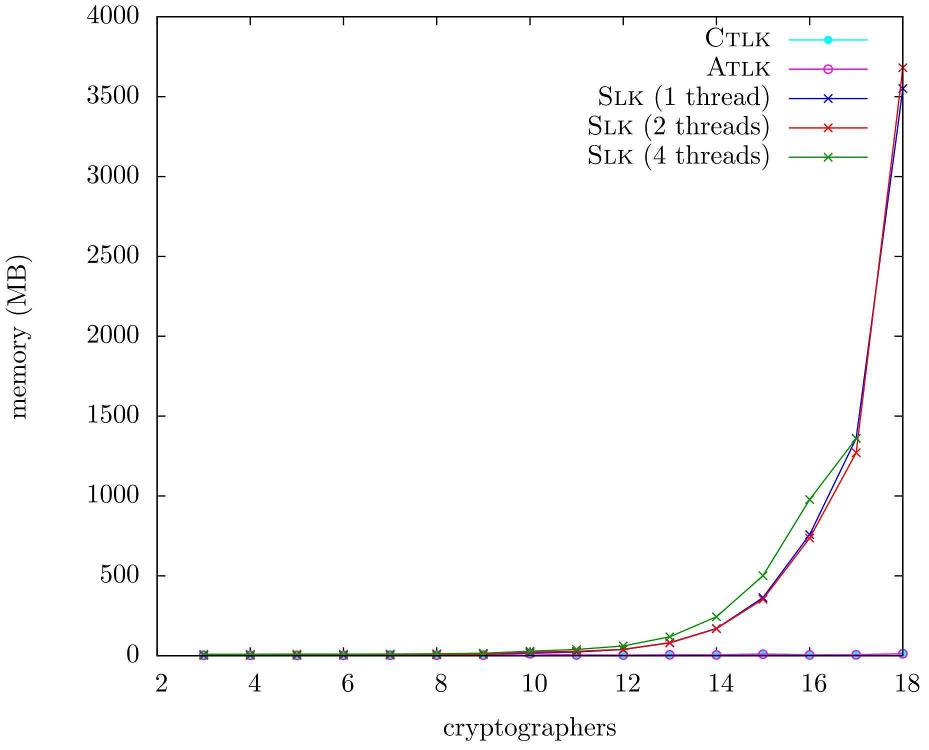
[8]We ran the experiment (with 4 threads and $n = 18$ cryptographers) repeatedly on different lab machines and it was killed each time by the operating system.

crypts n	**possible states**	**reachable states**	**reachability time** (s)	CTLK		ATLK		SLK					
								1 thread		2 threads		4 threads	
				time (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)	**mem** (MB)
3	82944	128	0.00	0.00	2.69	0.00	2.69	0.00	2.79	0.01	4.91	0.02	9.23
4	1.99×10^6	320	0.02	0.00	2.71	0.00	2.70	0.00	2.91	0.02	4.97	0.04	9.24
5	4.78×10^7	768	0.04	0.00	2.75	0.00	2.75	0.01	3.07	0.03	5.10	0.05	9.67
6	1.15×10^9	1792	0.08	0.00	2.79	0.00	2.77	0.02	3.41	0.04	5.37	0.07	9.94
7	2.75×10^{10}	4096	0.21	0.00	3.03	0.00	2.97	0.05	4.47	0.08	5.88	0.12	10.71
8	6.60×10^{11}	9216	0.40	0.00	3.19	0.00	3.27	0.18	6.37	0.21	7.42	0.24	12.56
9	1.58×10^{13}	20480	0.43	0.00	3.38	0.00	3.42	0.38	9.59	0.38	9.82	0.46	15.96
10	3.80×10^{14}	45056	4.19	0.28	13.02	0.16	12.75	2.29	18.62	2.23	22.33	2.43	28.21
11	9.13×10^{15}	98304	1.69	0.04	5.49	0.01	4.75	5.01	23.55	4.37	27.51	4.77	39.59
12	2.19×10^{17}	212992	2.32	0.01	4.40	0.01	4.20	11.61	39.25	9.61	40.36	10.32	60.82
13	5.26×10^{18}	458752	2.05	0.10	6.50	0.03	5.62	32.63	81.95	26.30	79.75	28.39	118.27
14	1.26×10^{20}	983040	1.96	0.08	6.87	0.03	5.00	89.46	169.95	67.99	169.68	73.57	243.80
15	3.03×10^{21}	2.10×10^6	21.35	0.35	12.73	0.08	9.29	163.77	363.66	139.42	354.18	156.57	500.96
16	7.27×10^{22}	4.46×10^6	6.66	0.09	6.65	0.04	4.83	422.03	758.43	345.83	736.02	377.50	977.79
17	1.74×10^{24}	9.44×10^6	9.10	0.13	6.67	0.08	6.04	734.44	1360.59	643.03	1270.93	705.44	1359.73
18	4.19×10^{25}	1.99×10^7	65.94	0.50	12.66	0.14	12.67	2654.46	3550.66	2129.54	3680.89	*process killed*	

Table 6.3: Experimental model checking results for the dining cryptographers protocol.



(a) Running time.



(b) Peak memory usage.

Figure 6.4: Experimental model checking results for the dining cryptographers protocol.

agents concurrently choose how they want to split the cake, while at odd rounds the environment decides how to actually slice the cake and assigns the obtained pieces to a subset of the agents. Therefore, the problem of splitting a cake of size d among n agents is divided into simpler problems in which cakes of size $d' < d$ have to be split between $n' < n$ agents. The game terminates once each agent receives a slice.

We model the game as an interpreted system (see Definition 2.5) with agents $Agt \triangleq \{\mathrm{E}, 1, \ldots, n\}$ where [26]:

- **Agents.** The set of *internal states* of an agent $i \in [1 \mathinner{..} n]$ is $L_i \triangleq [1 \mathinner{..} n] \times [1 \mathinner{..} d] \times \{\mathrm{P}, \mathrm{E}\}$, where the first component represents the number of agents $n' \leq n$ among which the current cake has to be split, the second component denotes the current size of the cake $d' \leq d$, and the flags P and E indicate whether it is the agents' or the environment's turn.

  The set of *actions* available to an agent $i \in [1 \mathinner{..} n]$ is $Act_i \triangleq \{\bot\} \cup [1 \mathinner{..} d]$ where $\bot$ indicates no choice (the agent already got his piece of cake or it is the environment's turn) and a numeric value represents the size of the slice he wants. The agent's *protocol* is defined as $P_i(\langle (m, k, \mathrm{E}), l_\mathrm{E} \rangle) \triangleq P_i(\langle (1, k, \mathrm{P}), l_\mathrm{E} \rangle) \triangleq \{\bot\}$ and $P_i(\langle (m, k, \mathrm{P}), l_\mathrm{E} \rangle) \triangleq [1 \mathinner{..} k]$ for $m > 1$.

- **Environment.** The internal states of the environment represent the possible divisions of the cake among subsets of agents. To represent them, we need to introduce some auxiliary concepts. Firstly, we define the set of functions $F \triangleq \{f : [1 \mathinner{..} n] \to [1 \mathinner{..} n] \mid \exists h \in [1 \mathinner{..} d] \,.\, \mathrm{img}(f) = [1 \mathinner{..} h]\}$ which assign agents to pieces of cake. Secondly, the sizes of the pieces are encoded by means of partial functions $S \triangleq [1 \mathinner{..} n] \rightharpoonup [1 \mathinner{..} d]$. The possible divisions of the cake are then pairs $(f, s) \in H \triangleq \left\{ (f, s) \in F \times S \;\middle|\; \mathrm{dom}(s) = \mathrm{img}(f) \wedge \sum_{i \in \mathrm{dom}(s)} s(i) \leq d \right\}$ where f maintains the information about the splitting whose size is contained in s.

  In order to define the protocol of the environment, we need to define a partial order on H which allows us to identify all possible continuations from a given state. For all $(f, s), (f', s') \in H$, we write that $(f', s') \preceq (f, s)$ iff the following two conditions hold:

  1. if $f'(i) = f'(j)$, then $f(i) = f(j)$, for all $i, j \in [1 \mathinner{..} n]$;
  2. $\sum_{j \in [1 \mathinner{..} n], f(j) = f(i)} s'(f'(j)) \leq s(f(i))$, for all $i \in [1 \mathinner{..} n]$.

  Intuitively, $(f', s') \preceq (f, s)$ if f' is a refinement of f, i.e. f' represents a finer partition than f, and s' is coherent with the sizes of the pieces s before the last cuts were made.

  At this point, we can define the *actions* of the environment $Act_\mathrm{E} \triangleq H$ and its *local states* $L_\mathrm{E} \triangleq Act = H \times \prod_{i=1}^{n} Act_i$. Intuitively, the Cartesian product stores the previous actions of agents $i \in [1 \mathinner{..} n]$ (so that the environment could base its decision on their actions). Finally, the protocol of the environment is defined as[9]:

  $$P_\mathrm{E}((a_\mathrm{E}, a_1, \ldots, a_n)) \triangleq \begin{cases} \{a_E\} & \text{if } a_i = \bot \text{ for all } i \in [1 \mathinner{..} n] \\ \{(f, s) \in H \mid (f, s) \preceq a_\mathrm{E}\} & \text{otherwise} \end{cases}$$

  Intuitively, when it is the turn of the players, the environment has to preserve the previous splitting. Otherwise, it can use all possible refinements on its previous move a_E.

- **Evolution.** Since the states of the environment maintain both the information on the splitting of the cake (environment's previous action) and the choices made by the agents in the previous round, the *environment evolution function* is simply $t_\mathrm{E}(a, a') \triangleq a'$.

  There are two cases for the *agents' evolution functions*:

  1. States of the form (m, k, P) (agents' turn) merely allow a subset of $m \leq n$ agents to make a proposal to the environment for the cut of their piece of cake of size $k \leq d$. Therefore, we have $t_i(\langle (m, k, \mathrm{P}), a \rangle, a') \triangleq (m, k, \mathrm{E})$ for all $i \in [1 \mathinner{..} n]$.

[9]The protocol might seem reversed because the environment makes no move when $a_i = \bot$ for all $i \in [1 \mathinner{..} n]$. However, it is important to realise that $a_1, \ldots, a_n$ refer to the *previous* actions of the agents. Hence, if the agents did not make any move in the previous round, it is now their turn (unless the game is over, in which case the environment should not do anything either).

2. States of the form (m, k, E) (environments' turn) are used to adopt the decision of the environment $a_{\mathrm{E}}(a) = (f, s)$. To do this, we set $t_i(\langle (m, k, \mathrm{E}), a\rangle, a') \triangleq (m', k', \mathrm{P})$ where $m' \triangleq |\{j \in [1 \mathinner{..} n] \mid f(j) = f(i)\}| \leq m$ and $k' \triangleq s(f(i))$ for all $i \in [1 \mathinner{..} n]$.

- **Initial states.** The set of initial states $I \triangleq \{g\}$ contains a unique global state $g \in G$ defined as:
  1. $l_i(g) = (n, d, \mathrm{P})$ for all $i \in [1 \mathinner{..} n]$;
  2. $l_{\mathrm{E}}(g) = ((f, s), \perp, \ldots, \perp)$ where $f(i) = 1$ for all $i \in [1 \mathinner{..} n]$ and $s(1) = d$.
- **Assignment.** We fix a set of *atomic propositions* $AP \triangleq [1 \mathinner{..} n] \times [1 \mathinner{..} d]$, where each atomic proposition $\langle i, c\rangle \in AP$ indicates that agent i got piece of cake of size c. The *assignment* $h : AP \to 2^G$ is formally defined as $h(\langle i, c\rangle) \triangleq \{g \in G \mid l_i(g) = (1, c, \mathrm{P})\}$ for all $\langle i, c\rangle \in AP$.

The existence of a protocol for the environment is then given by the following SLK specification:

$$\varphi \triangleq \langle\!\langle x \rangle\!\rangle(\varphi_{\mathrm{F}} \wedge \varphi_{\mathrm{S}}) \tag{6.2}$$

where:

- $\varphi_{\mathrm{F}} \triangleq [\![y_1]\!] \cdots [\![y_n]\!](\psi_{\mathrm{NE}} \to \psi_{\mathrm{E}})$ ensures that the protocol x is *fair*, i.e. all possible Nash equilibria $(y_1, \ldots, y_n)$ of the agents guarantee equality of splitting;
- $\varphi_{\mathrm{S}} \triangleq \langle\!\langle y_1 \rangle\!\rangle \cdots \langle\!\langle y_n \rangle\!\rangle \psi_{\mathrm{NE}}$ ensures that the protocol has a *solution*, i.e. there is at least one Nash equilibrium;
- $\psi_{\mathrm{NE}} \triangleq \bigwedge_{i=1}^{n} \left(\bigwedge_{v=1}^{d} (\langle\!\langle z \rangle\!\rangle \flat_i p_i(v)) \to \left(\bigvee_{c=v}^{d} \flat p_i(c)\right)\right)$ ensures that if agent i has a strategy z that allows him to obtain a piece of cake of size v once the strategies of the other agents are fixed, he is already able to obtain a slice of size $c \geq v$ by means of his original strategy y_i, i.e. the original strategies $y_1, \ldots, y_n$ form a *Nash equilibrium*;
- $\psi_{\mathrm{E}} \triangleq \flat \bigwedge_{i=1}^{n} p_i(\lfloor d/n \rfloor)$ ensures that agent i is able to obtain a piece of size $\lfloor d/n \rfloor$;
- $\flat \triangleq (\mathrm{E}, x)(1, y_1) \cdots (n, y_n)$, $\flat_i \triangleq (\mathrm{E}, x)(1, y_1) \cdots (i, z) \cdots (n, y_n)$, and $p_i(c) \triangleq \mathsf{F}\,\langle i, c\rangle$ are auxiliary abbreviations.

We were able to verify the formula φ defined above on a system with $n = 2$ agents and a cake of size $d = 2$. Moreover, the SLK extension can automatically synthesise a witness strategy h_x for the environment as well as the Nash equilibrium (h_{y_1}, h_{y_2}) of the agents[10]:

```
------- Strategy x [Environment] -------
Agent Environment
  E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=player (8): f_1_1_s_1
  E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=player (7): f_1_1_s_2
  E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=env (5): f_1_2_s_1_1
  E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=player (9): f_1_2_s_1_1
  E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=env (6): f_1_2_s_1_1
  E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=player (10): f_2_1_s_1_1
  E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=env (4): f_1_1_s_1
  E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env (3): f_1_1_s_2
  E.a1=1, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env (2): f_1_1_s_1
  E.a1=2, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env (1): f_2_1_s_1_1
  E.a1=2, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env (0): f_1_2_s_1_1
----------------------------------------
----------- Strategy y1 [P1] -----------
Agent P1
  E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=player, k=1, m=2 (8): a1
  E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=player, k=2, m=2 (7): a2
  E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=env, k=1, m=1 (5): a0
```

[10]`Environment` is abbreviated to `E`. A witness execution was also generated by the extension but we omit it for conciseness.

```
20    E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=player, k=1, m=1 (9): a0
21    E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=env, k=1, m=1 (6): a0
22    E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=player, k=1, m=1 (10): a0
23    E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=env, k=1, m=2 (4): a0
24    E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (3): a0
25    E.a1=1, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (2): a0
26    E.a1=2, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (1): a0
27    E.a1=2, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (0): a0
28  ----------------------------------------
29  ----------- Strategy y2 [P2] -----------
30  Agent P2
31    E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=player, k=1, m=2 (8): a1
32    E.a1=0, E.a2=0, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=player, k=2, m=2 (7): a1
33    E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=env, k=1, m=1 (5): a0
34    E.a1=0, E.a2=0, E.f1=1, E.f2=2, E.s1=1, E.s2=1, E.turn=player, k=1, m=1 (9): a0
35    E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=env, k=1, m=1 (6): a0
36    E.a1=0, E.a2=0, E.f1=2, E.f2=1, E.s1=1, E.s2=1, E.turn=player, k=1, m=1 (10): a0
37    E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=1, E.s2=0, E.turn=env, k=1, m=2 (4): a0
38    E.a1=1, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (3): a0
39    E.a1=1, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (2): a0
40    E.a1=2, E.a2=1, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (1): a0
41    E.a1=2, E.a2=2, E.f1=1, E.f2=1, E.s1=2, E.s2=0, E.turn=env, k=2, m=2 (0): a0
42  ----------------------------------------
```

For example, line 13 represents the following constraint on the strategy h_x for the environment:

$$h_x(((\overbrace{\{\underbrace{(1,1)}_{\texttt{E.f1=1}},\underbrace{(2,1)}_{\texttt{E.f2=1}}\}}^{f},\overbrace{\{\underbrace{(1,2)}_{\texttt{E.s1=2}}\}}^{s}),\overbrace{\underbrace{2}_{\texttt{E.a1=2}}}^{a_1},\overbrace{\underbrace{2}_{\texttt{E.a2=2}}}^{a_2})) = \underbrace{(\overbrace{\{(1,1),(2,2)\}}^{f'},\overbrace{\{(1,1),(2,1)\}}^{s'})}_{\texttt{f\_1\_2\_s\_1\_1}}$$

where $2 \notin \mathrm{dom}(s)$ is encoded by `E.s2=0`. Observe that $(f', s') \preccurlyeq (f, s)$ holds as required.

Unfortunately, we were not able to verify larger examples because the lab machine ran out of memory. For example, consider the scenario with $n = 2$ agents and $d = 3$ pieces of cake, which has 29 reachable states. The encoding of extended states, which contain strategies mapping each reachable state to an action, for the formula φ requires 105 Boolean variables, 54 of which represent the strategy h_x we wish to synthesise. The Boolean variables in turn generate a state space with 8.09×10^{20} possible *extended* states[11]. We found that the intermediate BDDs have order of 10^9 nodes, each of which has 32 bytes. A single intermediate BDD thus requires approximately 32GB of memory and the whole algorithm would use *several hundreds of gigabytes*.

This negative result should not be surprising given the theoretical difficulty of the cake-cutting problem. Moreover, we are synthesising the entire protocol and not just the agents' optimal behaviour. In our opinion, this is still a great achievement because our tool can automatically *synthesise Nash equilibria* of arbitrary memoryless games (given enough memory and time).

Note that the specification in Equation 6.2 again cannot be expressed in SL[1G]. Unlike the dining cryptographers specification in Equation 6.1, which contains epistemic modalities unsupported by SL[1G], this one cannot be translated to an equivalent SL[1G] formula because of the restrictions on syntax, namely that every quantification prefix $\wp$ must be coupled with a goal $\flat\varphi$ where $\flat$ binds all agents to variables in $\wp$ (see Definition 5.2).

[11]We were able to handle even larger state spaces (up to 4.19×10^{25} possible *global* states) in the dining cryptographers scenario (see Table 6.3). There are three important differences between the problems which we believe explain this discrepancy: *(i)* Equation 6.1 contains only one temporal operator (unlike Equation 6.2); *(ii)* the global state changes only during the first temporal transition in the dining cryptographers model, whereas the cake-cutting game might take several rounds; and *(iii)* unlike the environment protocol in the cake cutting problem, which we aim to synthesise, the dining cryptographers protocol is deterministic (once the coins are flipped).

6.4.3 Scheduler

Most operating systems nowadays support the concept of concurrency where several processes (or threads) run at the same time[12]. While parallelism can greatly improve the performance of many applications and simplify software design, it can also lead to race conditions where multiple processes use a shared resource simultaneously and its final state depends on the relative timing, which is non-deterministic and thus difficult to debug. Correctness of concurrent systems is crucial because they often run indefinitely and provide services that other systems depend on (e.g. web servers). Since humans generally tend to have difficulties reasoning about them and designing them, automatic verification and synthesis of such systems is of vital importance.

We consider a simple *preemptive scheduler system* [26] composed of $n \in \mathbb{N}$ processes and an arbiter (represented by the environment). The processes try to access a shared resource which is mutually exclusive. When more than one process requests the resource, the arbiter has to decide who will obtain it. We are interested in the following two properties:

1. **Mutual exclusion.** This property (also referred to as a *safety* property) asserts that at most one process owns the resource at any given point in time.

2. **Absence of starvation.** This property (also referred to as a *fairness* property) requires that every request is eventually satisfied, i.e. no process will wait for the resource forever.

Again, we model the scheduler system as an interpreted system (see Definition 2.5) with agents $Agt \triangleq \{\mathrm{E}, 1, \ldots, n\}$ where [26]:

- **Processes.** Every process $i \in [1 \mathinner{..} n]$ has three possible *internal states* $L_i \triangleq \{\mathrm{in}, \mathrm{wt}, \mathrm{rs}\}$ which stand for "internal computation", "waiting for resource", and "owns resource" respectively. Furthermore, the *actions* of the process are $Act_i \triangleq \{\mathrm{id}, \mathrm{rq}, \mathrm{rn}, \mathrm{rl}\}$ meaning "idle", "request resource", "relinquish resource", and "release resource" respectively. The third action ("relinquish resource") is used by the process to stop waiting for the resource. Finally, the *protocol* of the process is defined as $P_i(\mathrm{in}) \triangleq \{\mathrm{id}, \mathrm{rq}\}$, $P_i(\mathrm{wt}) \triangleq \{\mathrm{id}, \mathrm{rn}\}$, and $P_i(\mathrm{rs}) \triangleq \{\mathrm{id}, \mathrm{rs}\}$.

- **Arbiter.** A *state* of the environment either indicates that the resource is free, or represents the set of processes that require the resource but do not own it $L_{\mathrm{E}} \triangleq \{\mathrm{fr}\} \cup 2^{[1 \mathinner{..} n]}$. The *actions* of the environment are $Act_{\mathrm{E}} \triangleq \{\mathrm{id}\} \cup [1 \mathinner{..} n]$ where id indicates that the environment does not do any visible action (no arbitrage is required) and a numeric value $i \in [1 \mathinner{..} n]$ represents the fact that the arbiter has granted access to the shared resource to process i. The *protocol* of the environment is defined as $P_{\mathrm{E}}(\mathrm{fr}) \triangleq P_{\mathrm{E}}(\emptyset) \triangleq \{\mathrm{id}\}$ and $P_{\mathrm{E}}(A) \triangleq A$ for all $A \subseteq [1 \mathinner{..} n]$ with $A \neq \emptyset$.

- **Evolution.** Before formally defining the evolution functions, we introduce some auxiliary notation. Let $a \in Act$ be a joint action. The sets of agents that request, relinquish, and release the resource are defined as $\mathsf{Rq}(a) \triangleq \{i \in [1 \mathinner{..} n] \mid a_i(a) = \mathrm{rq}\}$, $\mathsf{Rn}(a) \triangleq \{i \in [1 \mathinner{..} n] \mid a_i(a) = \mathrm{rn}\}$, and $\mathsf{Rl}(a) \triangleq \{i \in [1 \mathinner{..} n] \mid a_i(a) = \mathrm{rl}\}$ respectively. Note that $|\mathsf{Rl}(a)| \leq 1$ for all possible joint actions $a \in Act$.

  There are three cases to consider for the *arbiter's evolution function*:

  1. Starting from the state fr, if no process makes a request, the state does not change. Otherwise, if there is exactly one process requesting the resource, the environment transits to $\emptyset$. In the case of concurrent requests, their set $\mathsf{Rq}(a)$ is recorded instead:

  $$t_{\mathrm{E}}(\mathrm{fr}, a) \triangleq \begin{cases} \mathrm{fr} & \text{if } |\mathsf{Rq}(a)| = 0 \\ \emptyset & \text{if } |\mathsf{Rq}(a)| = 1 \\ \mathsf{Rq}(a) & \text{otherwise} \end{cases}$$

  2. Starting from the state $\emptyset$, if the process that owns the resource releases it and no other process requests it, the environment transits to fr. If the owner does not release the resource and no

[12]Note that we do not care whether the system uses real or apparent concurrency. We are interested in its high-level behaviour instead.

process makes a request or if the owner releases it and exactly one process makes a request, the state does not changed. In all other cases, the set of concurrent requests $\mathsf{Rq}(a)$ is recorded:

$$t_{\mathrm{E}}(\emptyset, a) \triangleq \begin{cases} \mathrm{fr} & \text{if } |\mathsf{Rl}(a)| = 1 \text{ and } |\mathsf{Rq}(a)| = 0 \\ \emptyset & \text{if } |\mathsf{Rl}(a)| = |\mathsf{Rq}(a)| \\ \mathsf{Rq}(a) & \text{otherwise} \end{cases}$$

3. Finally, starting from a set $A \neq \emptyset$ of processes that are waiting for the resource, the environment transits to the new set $A \setminus (\{a_{\mathrm{E}}(a)\} \cup \mathsf{Rn}(a)) \cup \mathsf{Rq}(a)$ maintaining the set of processes that do not relinquish the request and are not chosen by the arbiter, augmented with the new ones that want to access the shared resource:

$$t_{\mathrm{E}}(A, a) \triangleq A \setminus (\{a_{\mathrm{E}}(a)\} \cup \mathsf{Rn}(a)) \cup \mathsf{Rq}(a)$$

The *evolution function of a process* $i \in [1 \mathinner{..} n]$ is defined analogously:

$$t_i(\langle \mathrm{in}, \mathrm{fr} \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{id} \\ \mathrm{rs} & \text{if } \mathsf{Rq}(a) = \{i\} \\ \mathrm{wt} & \text{otherwise} \end{cases}$$

$$t_i(\langle \mathrm{in}, \emptyset \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{id} \\ \mathrm{rs} & \text{if } |\mathsf{Rl}(a)| = 1 \text{ and } \mathsf{Rq}(a) = \{i\} \\ \mathrm{wt} & \text{otherwise} \end{cases}$$

$$t_i(\langle \mathrm{in}, A \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{id} \\ \mathrm{wt} & \text{otherwise} \end{cases}$$

$$t_i(\langle \mathrm{rs}, \emptyset \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{rl} \\ \mathrm{rs} & \text{otherwise} \end{cases}$$

$$t_i(\langle \mathrm{rs}, A \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{rl} \text{ or } a_{\mathrm{E}}(a) \neq i \\ \mathrm{rs} & \text{otherwise} \end{cases}$$

$$t_i(\langle \mathrm{wt}, A \rangle, a) \triangleq \begin{cases} \mathrm{in} & \text{if } a_i(a) = \mathrm{rn} \\ \mathrm{rs} & \text{if } a_i(a) = \mathrm{id} \text{ and } a_{\mathrm{E}}(a) = i \\ \mathrm{wt} & \text{otherwise} \end{cases}$$

- **Initial states.** The set of initial states $I \triangleq \{g\}$ contains a unique global state $g \in G$ such that $l_{\mathrm{E}}(g) = \mathrm{fr}$ and $l_i(g) = \mathrm{in}$ for all $i \in [1 \mathinner{..} n]$.
- **Assignment.** We fix a set of *atomic propositions* $AP \triangleq \{\mathrm{rs}, \mathrm{wt}\} \times [1 \mathinner{..} n]$, where $\langle \mathrm{wt}, i \rangle$ and $\langle \mathrm{rs}, i \rangle$ indicate that process $i \in [1 \mathinner{..} n]$ is waiting for and owns the resource respectively. The *assignment* $h : AP \to 2^G$ is formally defined as $h(\langle s, i \rangle) \triangleq \{g \in G \mid l_i(g) = s\}$ for all $i \in [1 \mathinner{..} n]$ and $s \in \{\mathrm{wt}, \mathrm{rs}\}$.

The desired properties of the system can be expressed using the following SL[1G][13] specifications:

1. **Mutual exclusion.** The following specification asserts that *all strategies* for the scheduler already ensure mutual exclusion, i.e. we want to *verify* the property:

$$\varphi_{\mathrm{ME}} \triangleq [\![x]\!][\![y_1]\!] \cdots [\![y_n]\!](\mathrm{E}, x)(1, y_1) \cdots (n, y_n)\, \mathsf{G} \neg \bigvee_{i=1}^{n} \bigvee_{j=i+1}^{n} \langle \mathrm{rs}, i \rangle \wedge \langle \mathrm{rs}, j \rangle$$

This property can be equivalently expressed in CTL and ATL using the AG and $\langle\!\langle \emptyset \rangle\!\rangle \mathsf{G}$ operators respectively (see Equation 6.1).

[13]Recall that SL[1G] syntax (Definition 5.2) is a subset of SLK syntax (Definition 4.1). Thus, SL[1G] specifications are also SLK specifications.

The model checking results for the formula with $2 \leq n \leq 10$ processes are shown in Table 6.4a. We can see that the Ctl and Atl formulas are checked almost instantaneously. Similarly to the dining cryptographers scenario (see Subsection 6.4.1), the Slk model checking performance on φ_{ME} drops quickly for $n \geq 8$ processes. However, the Sl[1g] model checking performance on φ_{ME} drops even faster and the procedure already takes more than 2 hours and 1.4GB memory on the model with $n = 7$ processes. Moreover, it runs out of physical memory (16GB) for $n > 7$ processes.

The individual steps of the Sl[1g] model checking algorithm (see Figure 5.3) which take more than one second, namely arena construction, parity game combination, and parity game solution, are also shown in Table 6.4a. We can see that the Sl[1g] model checking performance is dominated by parity game solving. For example, calculating the parity game solution for φ_{ME} with $n = 7$ processes takes approximately 1 hour and 50 minutes, which is roughly 77% of the whole model checking time (2 hours and 23 minutes). This is in line with our expectation that parity game solving is one of the bottlenecks of our Sl[1g] model checking algorithm.

2. **Absence of starvation.** The following specifications asserts the *existence* of a strategy for the scheduler which ensures absence of starvation, i.e. we want to *synthesise*[14] the strategy for:

$$\varphi_{\mathrm{AS}} \triangleq \langle\!\langle x \rangle\!\rangle [\![y_1]\!] \ldots [\![y_n]\!] (\mathrm{E}, x)(1, y_1) \cdots (n, y_n) \bigwedge_{i=1}^{n} \mathsf{G}\, (\langle \mathrm{wt}, i \rangle \rightarrow \mathsf{F} \neg \langle \mathrm{wt}, i \rangle)$$

This property cannot be expressed in Ctl because it refers to the capability of an agent to enforce a property. Moreover, it cannot be expressed in Atl either because it contains nested temporal operators (G and F) which depend on the same strategies. However, it can be translated to an Atl* formula by replacing the whole quantification and binding prefix with the quantifier $\langle\!\langle \{\mathrm{E}\} \rangle\!\rangle$.

We can use the equivalence $\bigwedge_{i=1}^{n} \mathsf{G}\, \psi_i \equiv \mathsf{G} \bigwedge_{i=1}^{n} \psi_i$ to reduce the number of temporal operators in φ_{AS} in order to improve Slk and Sl[1g] model checking performance. Since the underlying Ltl formula $\bigwedge_{i=1}^{n} \mathsf{G}\, \psi_i$ is a conjunction, we can also apply the *separate determinisation* optimisation technique for Sl[1g] discussed in Subsection 5.2.5. Therefore, we empirically evaluate the following three approaches: *(i)* the reduced formula with the Slk algorithm, *(ii)* the reduced formula with the unoptimised Sl[1g] algorithm, and *(iii)* the original formula with the optimised Sl[1g] algorithm.

The model checking results for the formulas with $2 \leq n \leq 3$ processes are shown in Table 6.4b. Unlike for φ_{ME}, Sl[1g] in this case outperforms Slk by an order of magnitude for $n \geq 2$ processes. Moreover, the Slk extension runs out of physical memory for $n > 3$ processes. We believe that the performance reversal reflects the different structures of φ_{ME} (safety condition) and φ_{AS} (fairness condition). A further investigation needs to be done in order to identify classes of formulas which can be checked efficiently by one of the two algorithms. The optimised Sl[1g] model checking algorithm is almost twice faster than the unoptimised one for $3 \leq n \leq 4$ processes and almost 5 times faster for $n = 5$ processes, i.e. the relative speedup increases with the number of processes. Moreover, the optimised algorithm can handle a system with $n = 6$ processes[15], whereas the unoptimised one runs out of memory. This is in line with our expectations (see Subsection 5.2.5).

Again, the individual steps of the Sl[1g] model checking algorithms (see Figures 5.3 and 5.4) which take more than one second, namely automaton determinisation, arena construction, parity game combination, and parity game solution, are shown in Table 6.4b. Similarly to φ_{ME}, Sl[1g] model checking performance is dominated by parity game solving. Unlike the optimised determinisation procedure, which is almost instantaneous, the unoptimised one is even slower than arena construction for $n \geq 4$ processes. This is due to the different sizes of the non-deterministic Büchi automata (see Subsection 5.2.5): The unoptimised algorithm determinises 1 automaton with

[14]Some of the Sl[1g] witness strategies for the formula have more than 10^5 states. Therefore, we do not actually request witness strategy synthesis (`-c 1` flag) because printing the strategies would become a major bottleneck of the algorithm which would skew the performance results. Omitting the flag does not have any impact on the performance of the algorithms themselves, which perform (symbolic) strategy synthesis internally anyway.

[15]Although it takes almost 24 hours.

$2^{2(n+1)+\lceil \log_2(n+2) \rceil}$ states[16], whereas the optimised one determinises n automata with 2^6 states[17]. This—together with the fact that the worst-case time complexity of the determinisation procedure is exponential with respect to the size of the non-deterministic Büchi automaton—explains why the optimised algorithm outperforms the unoptimised one and why the relative speedup increases with the number of processes.

6.4.4 Nim

Nim is a classical mathematical game for two players [77]:

> *"Nim is a two-player turn-based game in which players alternately remove objects from heaps. On each turn, one of the two players chooses a heap and a non-zero number of objects that he removes from this heap. The player who removes the last object wins the game."*

A complete theory for the game was developed already in 1901 [77]. Its key concept is the so-called *Nim-sum*, which is calculated as an exclusive or (xor) of the heap sizes. For example, if there are three heaps with sizes 3, 4, and 5 respectively, the Nim-sum is $011 \oplus 100 \oplus 101 = 010$. A game state is winning (for any of the two players) iff its Nim-sum is *not* 0. Consequently, the winning strategy is to finish every move with a Nim-sum of 0. In the previous example, the state is winning for the current player and his winning strategy is to remove 2 objects from the first heap.

We consider a variant of the game where an *upper bound* is imposed on the number of objects that can be removed in a turn[18]. We model the game with $n \in \mathbb{N}$ heaps with $k_1, \ldots, k_n \in \mathbb{N}$ elements and an upper bound $b > 0$ as an interpreted system (see Definition 2.5) with agents $Agt \triangleq \{\mathrm{E}, 1, 2\}$:

- **Environment.** The set of *internal states* of the environment is defined as $L_\mathrm{E} \triangleq \{\mathrm{t}_1, \mathrm{t}_2, \mathrm{w}_1, \mathrm{w}_2\} \times \prod_{j=1}^{n} [0 \mathinner{..} k_j]$ where t_1, t_2, w_1, and w_2 stand for "first player's turn", "second player's turn", "first player won", and "second player won" respectively. The environment has only one available *action* $Act_\mathrm{E} \triangleq \{\mathrm{id}\}$ meaning "idle". The environment *protocol* is simply $P_\mathrm{E}(l_\mathrm{E}) \triangleq \{\mathrm{id}\}$ for all $l_\mathrm{E} \in L_\mathrm{E}$.

- **Players.** Both players $i \in \{1, 2\}$ have only one available *internal state* $L_i \triangleq \{\mathrm{in}\}$, i.e. all information is in the internal state of the environment and thus both players have complete information (since their local states are $L_{i\mathrm{E}} \triangleq L_i \times L_\mathrm{E}$, see Definition 2.5). The set of actions available to each player is $Act_i \triangleq \{\mathrm{id}\} \cup \{\langle m, l \rangle \in \mathbb{N} \times \mathbb{N} \mid 1 \le m \le n \wedge 1 \le l \le \min(k_m, b)\}$ where id stands for "idle" and $\langle m, l \rangle$ indicates that the player removes l objects from heap m. The *protocol* of a player $i \in \{1, 2\}$ is defined as $P_i((\mathrm{in}, \langle \mathrm{w}_1, 0, \ldots, 0 \rangle)) \triangleq P_i((\mathrm{in}, \langle \mathrm{w}_2, 0, \ldots, 0 \rangle)) \triangleq P_i((\mathrm{in}, \langle \mathrm{t}_{1-i}, l_1, \ldots, l_n \rangle)) \triangleq \{\mathrm{id}\}$ and $P_i((\mathrm{in}, \langle \mathrm{t}_i, l_1, \ldots, l_n \rangle) \triangleq \{\langle m, l \rangle \in Act_i \setminus \{\mathrm{id}\} \mid l \le l_m\}$ for all $(l_1, \ldots, l_n) \in \prod_{j=1}^{n} [0 \mathinner{..} k_j] \setminus \{\langle 0, \ldots, 0 \rangle\}$.

- **Evolution.** The *environment evolution* is defined as follows. Firstly, if the a player $i \in \{1, 2\}$ has already won, the state does not change:

$$t_\mathrm{E}(\langle \mathrm{w}_i, 0, \ldots, 0 \rangle, a) \triangleq \langle \mathrm{w}_i, 0, \ldots, 0 \rangle$$

for all joint actions $a \in Act$. Secondly, if the next move of a player removes all objects from the last non-empty heap, the player wins:

$$t_\mathrm{E}(\langle \mathrm{t}_1, 0, \ldots, l_m, \ldots, 0 \rangle, (\mathrm{id}, \langle m, l_m \rangle, \mathrm{id})) \triangleq \langle \mathrm{w}_1, 0, \ldots, 0 \rangle$$
$$t_\mathrm{E}(\langle \mathrm{t}_2, 0, \ldots, l_m, \ldots, 0 \rangle, (\mathrm{id}, \mathrm{id}, \langle m, l_m \rangle)) \triangleq \langle \mathrm{w}_2, 0, \ldots, 0 \rangle$$

[16]Recall that a non-deterministic Büchi automaton $\mathfrak{B}$ equivalent to a non-deterministic generalised Büchi automaton $\mathfrak{A}$ is obtained as a product $\mathfrak{B} = \mathfrak{A} \times \mathfrak{D}$ with an auxiliary deterministic Büchi automaton $\mathfrak{D}$ with a counter (see the end of Subsection 2.4.1). In this particular case, $2(n+1)$ BDD variables represent the state of the underlying generalised non-deterministic Büchi automaton and $\lceil \log_2(n+2) \rceil$ variables represent the counter.

[17]4 BDD variables represent the state of the underlying generalised non-deterministic Büchi automaton and 2 variables represent the counter. Moreover, each of the n automata has only 6 reachable states which are not dead ends (recall that the number of BDD variables needed by the equivalent deterministic parity automaton depends on the number of reachable states which are not dead ends, see Subsection 2.4.4).

[18]This variant of Nim is referred to as *subtraction game* at `http://en.wikipedia.org/wiki/Nim`. According to the article, the winning regions of the game can be calculated using the Nim-sum of heap sizes modulo $b+1$ where $b > 0$ is the upper bound.

procs n	**possible states**	**reach. states**	**reach. time** (s)	Ctl		Atl		Slk		Sl[1G]				
				time (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)				**mem** (MB)
										total	arena	combine	solve	
2	72	9	0.00	0.00	2.64	0.00	2.64	0.00	2.67	0.01	0.00	0.00	0.01	2.99
3	432	21	0.00	0.00	2.72	0.00	2.72	0.00	2.81	0.14	0.02	0.03	0.09	5.23
4	2592	49	0.01	0.00	2.81	0.00	2.81	0.00	3.11	3.42	0.27	0.76	2.39	15.15
5	15552	113	0.02	0.00	3.02	0.00	3.03	0.02	4.20	65.81	2.82	11.58	51.40	33.37
6	93312	257	0.02	0.00	3.40	0.00	3.40	0.11	8.98	819.45	39.46	154.20	625.67	233.77
7	559872	577	0.08	0.00	4.47	0.00	4.48	0.86	23.18	8572.79	351.56	1597.84	6622.77	1500.69
8	3.36×10^6	1281	0.17	0.00	6.73	0.00	6.72	4.88	91.26	*out of memory*				
9	2.02×10^7	2817	0.20	0.00	11.62	0.00	11.62	36.71	561.69	*out of memory*				
10	1.21×10^8	6145	0.10	0.00	22.23	0.00	22.23	222.36	2769.64	*out of memory*				

(a) Mutual exclusion property φ_{ME}.

procs n	Slk		Sl[1G]											
			unoptimised						optimised					
	time (s)	**mem** (MB)	**time** (s)					**mem** (MB)	**time** (s)					**mem** (MB)
			total	determ.	arena	combine	solve		total	determ.	arena	combine	solve	
2	0.00	2.67	0.09	0.00	0.00	0.02	0.06	4.44	0.12	0.00	0.00	0.02	0.10	4.52
3	116.78	124.90	10.11	0.01	0.02	1.06	9.02	15.14	6.39	0.00	0.01	0.65	5.72	14.54
4	*out of memory*		631.11	0.33	0.26	86.47	544.04	41.80	338.16	0.00	0.24	23.33	314.57	40.70
5	*out of memory*		29593.56	5.86	2.61	4323.81	25261.15	2792.22	6131.43	0.00	2.65	444.06	5684.69	306.41
6	*out of memory*		*out of memory*						85976.57	0.00	38.27	8012.11	77925.96	2688.93

(b) Absence of starvation property φ_{AS}.

Table 6.4: Experimental model checking results for the preemptive scheduler system.

where $1 \leq m \leq n$ and $1 \leq l_m \leq \min(k_m, b)$. Finally, if none of the previous cases apply, the players' turns alternate:

$$t_{\mathrm{E}}(\langle \mathrm{t}_1, l_1, \ldots, l_m, \ldots, l_n \rangle, (\mathrm{id}, \langle m, l \rangle, \mathrm{id})) \triangleq \langle \mathrm{t}_2, l_1, \ldots, l_m - l, \ldots, l_n \rangle$$
$$t_{\mathrm{E}}(\langle \mathrm{t}_2, l_1, \ldots, l_m, \ldots, l_n \rangle, (\mathrm{id}, \mathrm{id}, \langle m, l \rangle)) \triangleq \langle \mathrm{t}_1, l_1, \ldots, l_m - l, \ldots, l_n \rangle$$

where $0 \leq l_j \leq k_j$ for all $j \in [1 \mathinner{..} n]$, $1 \leq m \leq n$, $1 \leq l \leq \min(l_m, b)$, and $(l_1, \ldots, l_m - l, \ldots l_n) \neq (0, \ldots, 0)$.

The *player evolution* is simply defined as $t_i(\mathrm{in}, a) \triangleq \mathrm{in}$ for both players $i \in \{1, 2\}$ and all joint actions $a \in Act$.

- **Initial states.** The set of initial states $I \triangleq \{g\}$ contains a unique global state $g \in G$ such that $l_{\mathrm{E}}(g) = \langle \mathrm{t}_1, k_1, \ldots, k_n \rangle$ and $l_1(g) = l_2(g) = \mathrm{in}$.
- **Assignment.** We fix a set of *atomic propositions* $AP \triangleq \{p_1, p_2\}$ where p_1 and p_2 mean that the first and the second player won respectively. The *assignment* is formally defined as $h(p_i) \triangleq \{g \in G \mid l_{\mathrm{E}}(g) = \langle \mathrm{w}_i, 0, \ldots, 0 \rangle\}$ for $i \in \{1, 2\}$.

The existence of a winning strategy for the first player is expressed in ATL and SL[1G] as:

$$\varphi_{\mathrm{ATL}} \triangleq \langle\!\langle \{1\} \rangle\!\rangle \mathsf{F}\, p_1 \qquad \varphi_{\mathrm{SL[1G]}} \triangleq \langle\!\langle s_1 \rangle\!\rangle [\![s_2]\!] [\![e]\!] (1, s_1)(2, s_2)(\mathrm{E}, e) \mathsf{F}\, p_1$$

Again, $\varphi_{\mathrm{SL[1G]}}$ is also an SLK formula. Note that the specification cannot be expressed in CTL because it refers to the capability of an agent to enforce a property.

The model checking results for the formulas on one heap ($n = 1$) of various sizes and an upper bound on the number of removed objects $b = 3$ are shown in Table 6.5. We can observe that ATL can easily handle heaps with 5×10^5 objects in less than 1 minute. Although the SL[1G] performance is worse (as expected given its model checking complexity), the algorithm can still model check heaps with 5×10^4 objects in a reasonable amount of time (less than one hour)[19]. This is mainly due to the fact that the specification has a fixed size. In contrast, SLK runs out of memory when there are more than 16 objects on the heap, i.e. its performance is several orders of magnitude worse than that of both ATL and SL[1G]. This might be surprising given that SLK outperformed SL[1G] on the mutual exclusion property in the scheduler scenario (see Table 6.4a). The reason is that in the Nim game, all agents have complete information. Consequently, SLK strategies map individual states to actions (rather than shared local states to actions as in the previous scenarios), i.e. different BDD variables must be allocated for every single reachable global state. Hence, incomplete information actually makes SLK model checking easier[20] because it reduces the number of BDD variables needed to encode strategies (see Subsection 4.2.4).

Again, the individual steps of the SL[1G] model checking algorithm (see Figure 5.3) which take more than one second, namely arena construction, parity game combination, and parity game solution, are shown in Table 6.5 as well. Unlike in the previous scenarios, arena construction takes a significant amount of time. In fact, it dominates SL[1G] performance when the heap has more than 2×10^4 objects. This is an unexpected result, which we suspect might be caused by the size of the intermediate BDDs and the operations of the underlying BDD package (e.g. variable reordering) triggered during the arena construction. Observe that, unlike SLK, SL[1G] has approximately the same memory usage as ATL.

6.4.5 Analysis

In general, both the SLK and the SL[1G] extension have significantly lower performance than the original tool on CTL and ATL. We expected this outcome given the high model checking complexities of the SL fragments. However, this does not mean that the extensions are not useful in practice. They can verify specifications which cannot be checked by the original tool. We have discussed two examples of such properties: *(i)* Nash equilibria (see Subsection 6.4.2) and *(ii)* fairness conditions (see Subsection 6.4.3). Moreover, both our extensions support witness strategy synthesis, which can be used to automatically

[19] Surprisingly, SL[1G] model checking of the heap with 2×10^4 objects takes less than half the time of model checking the heap with 10^5 objects. We believe this might be caused by the operations of the underlying BDD package (e.g. variable reordering) as we could not find any other explanation.

[20] Paradoxically, incomplete information makes SL[1G] model checking undecidable.

heap size	**possible states**	**reachable states**	**reachability time** (s)	Atl		Slk		Sl[1g]				
				time (s)	**mem** (MB)	**time** (s)	**mem** (MB)	**time** (s)				**mem** (MB)
								total	arena	combine	solve	
10	44	20	0.00	0.00	2.64	0.02	3.10	0.03	0.00	0.00	0.03	3.24
11	48	22	0.00	0.00	2.65	0.13	4.30	0.03	0.00	0.00	0.03	3.15
12	52	24	0.00	0.00	2.66	1.05	11.50	0.04	0.00	0.01	0.03	3.17
13	56	26	0.00	0.00	2.67	10.03	39.34	0.04	0.00	0.01	0.03	3.26
14	60	28	0.00	0.00	2.67	74.38	181.08	0.04	0.00	0.00	0.03	3.19
15	64	30	0.00	0.00	2.67	458.11	821.16	0.03	0.00	0.00	0.02	3.24
16	68	32	0.00	0.00	2.67	1423.07	1874.01	0.04	0.00	0.01	0.03	3.20
17	72	34	0.00	0.00	2.68	*out of memory*		0.04	0.00	0.01	0.03	3.26
18	76	36	0.00	0.00	2.68	*out of memory*		0.05	0.00	0.01	0.04	3.21
19	80	38	0.00	0.00	2.70	*out of memory*		0.04	0.00	0.00	0.04	3.20
20	84	40	0.00	0.00	2.71	*out of memory*		0.05	0.00	0.00	0.04	3.21
50	204	100	0.00	0.00	2.85	*out of memory*		0.09	0.00	0.00	0.08	3.36
100	404	200	0.00	0.00	3.06	*out of memory*		0.14	0.00	0.00	0.14	3.56
200	804	400	0.00	0.01	3.41	*out of memory*		0.31	0.00	0.00	0.30	3.95
500	2004	1000	0.01	0.01	4.55	*out of memory*		0.82	0.01	0.00	0.80	5.16
1000	4004	2000	0.01	0.03	6.13	*out of memory*		1.66	0.07	0.00	1.59	7.28
2000	8004	4000	0.03	0.06	9.23	*out of memory*		7.58	1.33	0.02	6.23	11.39
5000	20004	10000	0.08	0.16	19.27	*out of memory*		30.37	7.97	0.02	22.28	23.78
10000	40004	20000	0.20	1.39	35.31	*out of memory*		625.60	33.46	0.25	591.88	45.36
20000	80004	40000	0.44	0.89	67.93	*out of memory*		305.42	163.51	0.03	141.88	72.70
50000	200004	100000	1.12	2.62	160.76	*out of memory*		1894.03	1337.15	0.09	556.79	171.07
100000	400004	200000	2.26	3.56	312.25	*out of memory*		7606.22	7152.87	0.01	453.34	324.56
200000	800004	400000	39.62	59.35	617.69	*out of memory*		59235.59	32838.33	4.82	26392.43	649.08
500000	2000004	1.00×10^6	43.99	57.43	1538.71	*out of memory*		*runs for more than 48 hours*				

Table 6.5: Experimental model checking results for the Nim game with one heap ($n = 1$) and at most 3 objects removed every turn ($b = 3$).

generate agents' behaviour that ensures the properties. In other words, the aim of the extensions is not to replace the existing tool, but to add support for completely new features. Therefore, the results presented in this section should be regarded as an illustration of the new functionality (rather than a benchmark with negative performance results).

The experimental results presented in this section confirm that the main performance bottleneck of the Slk model checking algorithm is the BDD encoding of the extended states, which allocates separate BDD variables for each shared local state of a strategy (see Subsection 4.2.4). As we have seen in the Nim scenario (see Subsection 6.4.4), this problem is more pronounced when all agents have complete information, in which case shared local states collapse to reachable global states. Informally, the less information agents have the better the Slk model checking performance is. This is quite a rare situation because incomplete information usually leads to higher model checking complexity (Atl with imperfect recall [23]) or even undecidability (Atl with perfect recall [35]). Nevertheless, the Slk extension can handle reasonably large state spaces for certain specifications as we have seen in the dining cryptographers scenario (see Subsection 6.4.1). Its performance can be slightly improved by using multiple threads (see Table 6.3). However, as we discussed at the end of Subsection 6.2.3, the potential of this approach seems to be limited because the CUDD BDD package used by MCMAS has no built-in support for concurrency.

The performance of the Sl[1g] extension depends strongly on the specification. This is a direct consequence of the high Sl[1g] model checking complexity (2ExpTime-complete with respect to the size of the formula). As we have seen in the Nim scenario (see Subsection 6.4.4), the Sl[1g] extension performs very well on large state spaces when the formula has a fixed size (see Table 6.5). Furthermore, the optimisation technique discussed in Subsection 5.2.5 significantly improves the model checking performance of Sl[1g] conjunctions (see Table 6.4b). The relative performance of the Slk and Sl[1g] extension also depends strongly on the type of the formula (compare Tables 6.4a and 6.4b). We found that the main bottlenecks of the Sl[1g] model checking algorithms (see Figures 5.3 and 5.4) are parity game solving and arena construction. While the former is in line with our expectations, the reason for the latter remains unclear (see the discussion at the end of Subsection 6.4.4).

6.5 Summary

In this chapter, we discussed the existing MCMAS model checker and our two extensions for it which implement the novel model checking algorithms for Slk and Sl[1g] introduced in Chapters 4 and 5 respectively. The experimental results we obtained on several scalable scenarios demonstrate the feasibility of practical model checking and strategy synthesis for both Sl fragments despite the high theoretical complexities.

The first part of the chapter gave an overview of the existing tool and our two extensions. We started by describing the functionality, usage, and underlying architecture of the original tool. We then explained how each extension modified it in order to add support for the new features. We also demonstrated the ability of the extensions to synthesise strategies for Slk and Sl[1g] specifications. Furthermore, we described an experimental parallel implementation of the Slk extension and discussed its limitations caused by the fact that the CUDD BDD package used by MCMAS does not support concurrency.

The second part of the chapter focused on the experimental results of the extensions. We introduced several scalable scenarios which we used to compare the performance of the original tool on Ctl and Atl with the performance of the extensions on Slk and Sl[1g]. The scenarios also demonstrated the greater expressiveness of the Sl fragments (compared to Ctl and Atl) as well as their relative performance, which strongly depends on the formula to be verified. We found that although the extensions have lower performance than the original tool on Ctl and Atl, they are able to handle reasonably large state spaces.

Chapter 7

Evaluation

In Chapters 4 and 5, we introduced novel model checking algorithms for Epistemic Strategy Logic (Slk) and One-Goal Strategy Logic (Sl[1g]) respectively. Furthermore, in Chapter 6, we presented two extensions of the existing model checker MCMAS (see Subsection 2.5.2) which implement the new model checking algorithms. In this chapter, we evaluate both the algorithms and their implementation. We then summarise the differences between the two Sl fragments.

7.1 Theory

The first part of this project involved designing novel model checking algorithms for fragments of Sl which can be implemented symbolically. We have done the following:

1. **Epistemic Strategy Logic** (Chapter 4)**.** We defined a new fragment of Sl, Epistemic Strategy Logic (Slk), on imperfect recall semantics with incomplete information. Slk is a specification language for multi-agent systems which combines Ltl temporal connectives, Sl strategy operators, and epistemic modalities. On the one hand, this combination results in a very powerful formalism for reasoning about game-theoretic concepts as well as agents' knowledge. On the other hand, Slk specifications suffer from certain limitations due to memoryless strategies, as discussed in Subsection 4.1.5.

   We have defined the Slk model checking problem and proved that it is in the PSpace complexity class with respect to both the size of the model and the formula (see Theorem 4.1). Furthermore, we have provided a model checking algorithm for Slk (see Subsection 4.2.2) which admits an efficient symbolic implementation (see Subsection 4.2.4). Finally, we discussed the problem of general Slk strategy synthesis and provided a procedure for constructing witness and counterexample strategies (see Subsection 4.2.3).

2. **One-Goal Strategy Logic** (Chapter 5)**.** We redefined the syntax and semantics of an existing fragment of Sl, One-Goal Strategy Logic (Sl[1g]), on interpreted systems (see Definition 2.5). Despite being the least expressive syntactic fragment of full Sl, Sl[1g] subsumes many well-established temporal logics including those in the Atl* hierarchy while still having the same model checking complexity as Atl*, namely 2ExpTime-complete with respect to the size of the specification and P-complete with respect to the size of the model.

   We provided a novel procedure which reduces Sl[1g] model checking to the problem of solving a parity game (see Subsection 5.2.1). We proved its correctness and showed that it is optimal in the sense that it achieves the lower complexity bound (see Theorem 5.2). Moreover, we provided an efficient symbolic implementation of the procedure with the same time complexity (see Subsection 5.2.4). Finally, we have explained how it can be extended to support general strategy synthesis for arbitrary Sl[1g] formulas (see Subsection 5.2.3).

   In addition, we proposed an optimisation technique for the Sl[1g] model checking algorithm called *separate determinisation* (see Subsection 5.2.5) and described its impact on the complexity. We also explained how the symbolic implementation can be modified to incorporate the optimisation.

We shall now discuss the strengths and weaknesses of our theoretical contributions.

7.1.1 Strengths

Our theoretical contributions possess the following strengths:

1. **First practical** SL **algorithm.** All algorithms for model checking SL (or its fragments) introduced so far are very theoretical. They are mostly intended to serve as proofs of model checking decidability rather than to be implemented and used in practice. In contrast, our novel SLK and SL[1G] model checking algorithms were designed with implementation in mind from the very beginning. Furthermore, they both admit efficient symbolic representations (see Subsections 4.2.4 and 5.2.4). In our opinion, this is the first step towards practical SL model checking.

2. **Optimal** SL[1G] **complexity.** Not only have we designed the first practical algorithm for model checking SL[1G], but the algorithm also has optimal SL[1G] model checking time complexity, i.e. its running time is polynomial in the size of the model and doubly-exponential in the size of the specification (see Subsection 5.2.2). Moreover, it is optimal for ATL* model checking as well because ATL* has the same model checking complexity as SL[1G] (despite being strictly less expressive). However, the algorithm does not have optimal complexity for all logics in the ATL* hierarchy (e.g. LTL and CTL* are PSPACE-COMPLETE with respect to the size of the formula).

3. **Symbolic implementation.** Our model checking and strategy synthesis algorithms for both SLK and SL[1G] can be implemented symbolically using BDDs (see Subsections 2.3.1 and 2.3.2). While the theoretical complexity is not improved by the data structure[1], BDDs generally outperform explicit approaches in practice by several orders of magnitude (see Table 2.2). This is especially important because of the high model checking complexity of both fragments.

4. **Modularity.** Our novel model checking algorithms for SLK and SL[1G] are highly modular in the sense that individual steps can be replaced by alternative procedures with minimal impact on the rest of the algorithm. For example, the method for translating a non-deterministic Büchi automaton to a deterministic parity automaton (see Subsection 2.4.4) could be replaced with a different determinisation procedure by changing only one line of the SL[1G] model checking algorithm. We hope that our algorithm (or some of its parts) might serve as a basis for model checking algorithms for other SL fragments like Disjunctive/Conjunctive-Goal Strategy Logic (SL[DG/CG], see Section 3.2).

5. **Formally correct.** All concepts used in our model checking and strategy synthesis algorithms are formally defined. Moreover, all steps of the algorithms either reuse existing techniques (e.g. standard translation in Figure 2.6) or apply novel procedures which we proved are correct (e.g. combined parity game equivalence in Theorem 5.1). Therefore, our solutions of the model checking and strategy synthesis problems for SLK and SL[1G] are both sound and complete.

The main strength of our novel model checking algorithms for SLK and SL[1G] is the fact that they can be implemented symbolically. Moreover, we proved that both algorithms are correct and that the worst-case time complexity of our SL[1G] model checking algorithm is optimal.

7.1.2 Weaknesses

The main weakness of our theoretical contributions is the *suboptimal complexity* of the SLK model checking algorithm. We have shown that the SLK model checking problem belongs to the PSPACE class in Subsection 4.2.1. However, as we have already pointed out, the symbolic implementation of the SLK model checking algorithm presented in Subsection 4.2.4 might use more than a polynomial amount of space. Therefore, our SLK algorithm does not have optimal space complexity. Instead, we argued that it has singly-exponential worst-case time complexity in both the size of the model and the formula. We strongly believe that the space suboptimality is outweighed by the empirical efficiency of binary decision diagrams (see point 3 in Subsection 7.1.1).

[1]In fact, our symbolic implementation of the SLK model checking algorithm has suboptimal space complexity (see Subsection 7.1.2).

7.2 Implementation

The second part of our project involved implementing the novel model checking algorithms and testing their performance. We have done the following:

1. **Implementation** (Sections 6.2 and 6.3)**.** We implemented our model checking algorithms for SLK and SL[1G] as extensions of the existing model checker MCMAS (see Subsection 2.5.2). In both cases, the existing ISPL syntax was extended to accommodate for the new operators and the algorithm was translated to a sequence of symbolic operations on binary decision diagrams (see Subsections 2.3.1 and 2.3.2). In addition, the enhanced tool adds support for SLK witness/counterexample strategy synthesis (see Subsection 6.2.2) and SL[1G] witness/counterexample and general strategy synthesis (see Subsection 6.3.2).

   In addition, we provided an *experimental parallel implementation* of the SLK model checking algorithm (see Subsection 6.2.3). The separate determinisation optimisation technique (see Subsection 5.2.5) is also supported by the SL[1G] extension.

2. **Experimental results** (Section 6.4)**.** We tested both extensions on several scalable real-life scenarios and compared their performance with the performance of MCMAS on equivalent CTL and ATL formulas (see Tables 6.3, 6.4a, and 6.5). We also demonstrated the expressiveness of the fragments together with witness strategy synthesis (see Subsection 6.4.2). Finally, we compared the relative performance of the two SL fragments and evaluated the impact of the proposed optimisation techniques (experimental parallel implementation for SLK and separate determinisation for SL[1G]).

We shall now discuss the strengths and weaknesses of our implementation.

7.2.1 Strengths

Our implementation possess the following strengths:

1. **First SL tool.** As we have already mentioned several times, we are not aware of any existing tool that would support SL or any of its fragments. Therefore, MCMAS (with our extensions) is the first tool that supports SLK and SL[1G]. In our opinion, this is the biggest strength of our project because it demonstrates that model checking and strategy synthesis for both fragments is feasible in practice.

   In fact, we are not even aware of any existing tool that would support ATL* (see Subsection 2.2.4). Therefore, we believe that our product is the first tool to support[2] ATL*. Consequently, MCMAS can now handle all logics in the ATL* hierarchy which it did not support before including LTL and CTL* (see Subsections 2.2.1 and 2.2.3).

2. **Symbolic implementation.** Since our novel model checking algorithms for SLK and SL[1G] admit efficient symbolic representations (see item 3 in Subsection 7.1.1), the extensions implement them using operations on BDDs (see Subsections 2.3.1 and 2.3.2), which outperform explicit approaches in practice by several orders of magnitude (see Table 2.1).

3. **Modularity.** Similarly to the new model checking algorithms (see item 4 in Subsection 7.1.1), our implementation of both MCMAS extensions is highly modular. This is very good from a software engineering perspective because our code can be easily reused or modified in the future by other people.

4. **Ease of use.** Despite its wide range of functionality, MCMAS is a simple command-line tool which is easy to use. The user only has to specify the name of the ISPL file to check and possibly pass a command-line flag if extra functionality is required (e.g. counterexample/witness generation). Both the input (ISPL syntax) and output of the tool is mostly self-explanatory and therefore simple to understand.

[2]As we have already explained in Subsection 6.3.1, the ATL* formula needs to be rewritten to an equivalent SL[1G] formula first. However, this is a very straightforward translation, which could be automatised by extending the ISPL syntax with ATL* strategy quantifiers.

In terms of input, our extensions merely *(i)* add three new command-line flags (listed in MCMAS help) and *(ii)* naturally extend the syntax with SL strategy operators (see Table 6.2). We believe that the format of the output is also quite easy to understand: SLK strategies are presented directly as mappings from states to actions and SL[1G] strategies are represented as automata (see Subsections 6.2.2 and 6.3.2).

The main strength of our solution is its novelty. We developed the first tool which supports model checking and strategy synthesis for fragments of SL. Moreover, the underlying algorithms for both fragments are implemented symbolically using efficient operations on BDDs.

7.2.2 Weaknesses

Our implementation has the following weaknesses:

1. **Performance.** The experimental results presented in Section 6.4 indicate that the biggest weakness of both extensions is their relatively low performance (compared to the performance of the original tool on CTL and ATL). This should not be surprising given the high expressiveness and therefore high model checking complexity of both SLK and SL[1G]:

   While CTL and ATL belong to the P complexity class (see Table 2.1), the SLK and SL[1G] model checking problems are PSPACE[3] and 2EXPTIME-COMPLETE respectively with respect the size of the *specification*. For this reason, the tool cannot handle large SLK and SL[1G] formulas in general. Nevertheless, the performance of the extensions can be highly optimised for certain types of formulas: We implemented an optimisation technique for SL[1G] model checking called *separate determinisation* that speeds up the model checking of conjunctions of LTL formulas, which occur often in practice (see Subsection 5.2.5). As we can see in Table 6.4b, the technique significantly reduces both model checking time and memory usage and thereby increases the size of the state space that the extension can handle. Other possible optimisation techniques are discussed in Section 8.2.

   Although our SLK model checking algorithm has exponentially lower time complexity with respect to the size of the formula than the one for SL[1G], it appears to be less feasible in practice. The time complexity of the model checking algorithm with respect to the size of the *model* is much more important because specifications are usually reasonably concise even for large state spaces. This turns out to be the main problem of our SLK model checking algorithm, which cannot handle large state spaces because it runs in (worst-case) exponential time in the size of the model (unlike the algorithms for CTL, ATL, and SL[1G]). This problem manifested itself in the Nim game (see Subsection 6.4.4), where the SLK performance was several orders of magnitude worse than that of both ATL and SL[1G]. Unfortunately, it appears that this is an unavoidable consequence of imperfect recall semantics (rather than a shortcoming of our algorithm).

2. **Limited testing.** As we have discussed in Subsection 6.1.3, the source code of MCMAS has a very bad design from a software engineering perspective (e.g. it has no tests whatsoever). While we tried to apply object-oriented programming principles during the development of the extensions (e.g. we used class hierarchy to represent different types of ω-automata as shown in Figure 6.3), we had to reuse some existing patterns in order to avoid "reinventing the wheel". In particular, our code relies on global structures storing the BDD parameters, which makes it extremely difficult to write unit tests for individual procedures.

   We have partially mitigated this issue by developing a framework for *system testing* of MCMAS and creating a test suite with several pairs of sample input and output files instead. Both model checking and strategy synthesis (for the new SL fragments) were tested by the framework. However, we think that unit tests for all MCMAS functions (not only our extensions) should be written if the tool is ever redesigned in the future and the dependency on global variables is reduced.

Overall, the main weakness of the implementation is its low performance, which is not surprising given the novelty of the algorithms and the high complexity of SLK and SL[1G] model checking. This issue can be mitigated to a large extent by applying optimisation techniques for various types of specifications. The development of such techniques is one of the most promising future directions of this project (see Section 8.2).

[3]As we have explained in Subsection 4.2.1, we conjecture that the SLK model checking problem is also PSPACE-HARD.

7.3 Fragment Comparison

Throughout this project, we were developing model checking frameworks for two fragments of SL in parallel. Our choice of the fragments, namely SLK and SL[1G], was explained in Chapter 3. The relevant theory and our novel model checking algorithms for both logics were presented in Chapters 4 and 5 respectively. The implementations of the algorithms in the form of MCMAS extensions were discussed in Sections 6.2 and 6.3 respectively. Finally, the experimental results of both extensions were provided in Section 6.4:

	SLK	SL[1G]
Fragment selection	Chapter 3	
Theory	Section 4.1	Section 5.1
Model checking	Section 4.2	Section 5.2
Implementation	Section 6.2	Section 6.3
Experimental results	Section 6.4	

We shall now summarise the differences and similarities between the two fragments as well as the corresponding model checking algorithms, implementations, and experimental results:

1. **Syntax.** Both fragments use SL strategy operators ($\langle\langle x\rangle\rangle$, $[\![x]\!]$, (i, x) with $x \in \mathit{Var}$ and $i \in \mathit{Agt}$, see Definition 2.19) and LTL temporal operators (X, F, G, U, see 2.7). While there are no restrictions[4] on the structure of SLK formulas (e.g. an agent can be bound to a different strategy while the strategies of the other agents remain the same, see Definition 4.1), SL[1G] quantification and binding prefixes must be coupled together (i.e. all agents are always bound to newly quantified strategies simultaneously, see Definition 5.2). Furthermore, SLK adds support for epistemic modalities expressing agents knowledge (K_i E_A, D_A, C_A with $i \in \mathit{Agt}$ and $A \subseteq \mathit{Agt}$, see Subsection 2.2.6), which SL[1G] does not support because its semantics is defined with respect to complete information. Therefore, all SL[1G] formulas are also SLK formulas (but not the other way round), i.e. *SL[1G]* $\subset$ *SLK*.

2. **Semantics.** We defined the semantics of both fragments on interpreted systems (see Definition 2.5). SLK uses *imperfect recall semantics with incomplete information* (i.e. agents have no memory of the past and do not have complete knowledge of the system, see Definition 4.3), whereas SL[1G] uses *perfect recall semantics with complete information* (i.e. agents remember the whole history and have complete knowledge of the system, see Definition 2.29). As we explained in Chapter 3, we considered these two semantics because perfect recall semantics with incomplete information—which we would have ideally used instead—leads to an undecidable model checking problem.

3. **Strategies.** A key concept of both fragments are strategies which map possible configurations of the system to actions. The types of strategies that both fragments use are closely connected to their semantics (item 2):

   SLK uses *uniform shared memoryless strategies* (see Definition 4.2), which map shared local states (of a group of agents $A \subseteq \mathit{Agt}$) to actions, i.e. $f_A : G/{\sim_A^{\mathsf{C}}} \to \mathit{Act}_A$. SLK strategies are *non-behavioural*, i.e. the next action of an agent in one state might depend on the actions of other agents in counterfactual evolutions of the system (see Subsection 4.1.5).

   SL[1G] uses *shared memoryful strategies* (see Definition 2.21), which map non-empty finite sequences of global states (tracks) to actions, i.e. $f_A : \mathit{Trk} \to \mathit{Act}_A$. Unlike SLK strategies, SL[1G] strategies are generally *not uniform* because a single local state history of an agent might be mapped to two different actions. However, SL[1G] strategies are *behavioural* (see Subsection 5.2.1).

4. **Expressiveness.** Both fragments can express the ability of agents to enforce temporal properties of the system by means of *shared interdependent* strategies (e.g. *"For all strategies of agent a, there exists a shared strategy for agents b and c, such that for all strategies ..."*). Therefore, SLK and SL[1G] are more expressive[5] than ATL* with imperfect and perfect recall respectively, which only

[4]As long as the formulas are *sentences* (see Definition 2.20).

[5]Recall that there is a subtle difference between SLK and ATL* semantics in the way they handle universally quantified agents (see Subsection 4.2.1). Therefore, strictly speaking, SLK does *not* subsume ATL* with imperfect recall. However, SL[1G] still strictly subsumes ATL* with perfect recall (despite the difference).

Logic		Complexity w.r.t. $\lvert\varphi\rvert$	Complexity w.r.t. $\lvert\mathcal{I}\rvert$
imperfect recall	ATL*	PSPACE-COMPLETE	PSPACE-COMPLETE
	SLK	PSPACE	PSPACE
perfect recall	ATL*	2EXPTIME-COMPLETE	P-COMPLETE
	SL[1G]	2EXPTIME-COMPLETE	P-COMPLETE

Table 7.1: Model checking problem complexity of ATL*, SLK (see Theorem 4.1), and SL[1G] with respect to the size of the model $\lvert\mathcal{I}\rvert$ and the formula $\lvert\varphi\rvert$ [23, 72]. We conjecture that SLK is PSPACE-COMPLETE with respect to both the size of the model and the formula (see Subsection 4.2.1).

refer to *individual independent* strategies (e.g. *"There exists a strategy for agent d, such that no matter what the other agents do, ..."*). This difference is due to the fact that SL handles strategies explicity via quantifiers, whereas ATL* strategies are only implicit.

Since SLK syntax is not restricted and supports epistemic modalities, SLK is more expressive[6] than SL[1G]: As we have seen in the dining cryptographers and scheduler scenarios (see Subsections 6.4.1 and 6.4.3), SLK can express agents' knowledge and game-theoretic concepts like Nash equilibria. We explained that neither of these properties can be expressed in SL[1G].

5. **Model checking *problem* complexity.** The theoretical model checking problem complexity of both SLK and SL[1G] as well as ATL* is shown in Table 7.1. We can see that the SL fragments have the same worst-case complexities as their ATL* counterparts despite being more expressive. Moreover, SL[1G] has exactly the same model checking complexity as ATL* with perfect recall, which it strictly subsumes. As we have explained in Subsection 4.2.1, we conjecture that SLK has the same model checking complexity as its ATL* counterpart as well.

   Both SL fragments have high model checking complexities, which have a negative impact on the performance of our extensions (see Section 6.4).

6. **Model checking *algorithm* complexity.** The worst-case time complexity of our SL[1G] model checking algorithm is *optimal* in the sense that it achieves the lower bounds on the problem complexity, i.e. doubly-exponential in the size of the formula and polynomial in the size of the model (see Theorems 5.2 and 5.4):

$$\lvert\mathcal{I}\rvert^{2^{O(\lvert\varphi\rvert)}}$$

   Slightly tighter bounds for the unoptimised and optimised SL[1G] algorithm are provided in Subsections 5.2.2 and 5.2.5 respectively. Moreover, we argued that the the algorithm is also optimal for ATL* (see item 2 in Subsection 7.1.1).

   The worst-case time complexity of our SLK algorithm is exponential in both the size of the model and the formula (see Theorem 4.3):

$$2^{O(\lvert\mathcal{I}\rvert\times\lvert\varphi\rvert)}$$

   While we are uncertain whether a sub-exponential time complexity can be achieved (in which case the time complexity of our algorithm would be suboptimal), we know that the space complexity of our algorithm is *suboptimal* because the problem belongs to the PSPACE complexity class (see Theorem 4.1), but our algorithm may use more than a polynomial amount of space (see Subsection 4.2.4).

7. **Optimisation techniques.** We have designed optimisation techniques for both SL fragments in order to improve the performance of the algorithms and extensions:

   We developed an *experimental parallel implementation* of our SLK extension which uses multiple threads to enumerate the local states of an agent (see Subsection 6.2.3). Although it does have an impact on SLK performance (two threads reduced model checking time by roughly 16% in the dining cryptographers scenario, see Table 6.3 and Figure 6.4), it is not very scalable because the CUDD BDD package used by MCMAS has no built-in support for concurrency.

[6]This does *not* mean that SLK subsumes SL[1G]. Recall that there is a fundamental difference between the semantics of the fragments: SLK is defined with respect to *memoryless* strategies, whereas SL[1G] is defined with respect to *memoryful* strategies.

We proposed an optimisation technique for the SL[1G] model checking algorithm called *separate determinisation* (see Subsection 5.2.5), which improves the performance of the algorithm on SL[1G] principal sentences containing conjunctions of LTL formulas, i.e. SL[1G] sentences of the form $\wp\flat \bigwedge_{i=1}^{n} \psi_i$. Our experimental results indicate that the optimisation has a major impact on model checking time and memory: It reduced the time to verify an SL[1G] formula with 5 conjuncts on a model with 1.56×10^4 states almost 5 times and increased the state space that the SL[1G] extension can handle (see Table 6.4b). Moreover, the positive effect of the optimisation appears to increase with the number of conjuncts.

8. **Strategy synthesis.** Both extensions support *witness/counterexample strategy synthesis* for formulas of the form $\langle\!\langle x_0 \rangle\!\rangle \cdots \langle\!\langle x_{m-1} \rangle\!\rangle \psi$ and $[\![y_0]\!] \cdots [\![y_{n-1}]\!] \phi$ (see Subsections 6.2.1 and 6.3.1). In addition, the SL[1G] extension supports *general strategy synthesis* for arbitrary SL[1G] sentences, which is demonstrated in Subsection 6.3.2. This functionality is not provided by the SLK extension because SLK strategies are non-behavioural, which makes general strategy synthesis extremely difficult as explained in Subsection 4.2.3.

9. **Command-line flags.** Both extensions reuse the existing `-c` ***`FORMAT`*** flag for enabling witness/counterexample strategy synthesis (see Subsections 6.2.2 and 6.3.2). Note that the extensions ignore the ***`FORMAT`*** parameter, which takes values `1`–`3`. In addition, they introduce extra command-line flags specific to their functionality:

   The SLK extension adds the `-t` ***`DEPTH`*** flag which enables thread branching of the experimental parallel implementation (with a given depth).

   The SL[1G] extension adds the `-solutions` and `-sd` ***`SWITCH`*** flags which enable general strategy synthesis and toggle separate determinisation respectively (the ***`SWITCH`*** parameter can take values `0` and `1`).

10. **Performance.** The performance of both extensions is relatively low compared to that of MCMAS on CTL and ATL (see item 1 in Subsection 7.2.2). This is not surprising given the high model checking complexities of the two SL fragments. We found that their performance depends on different factors: SLK performance depends mainly on the size of the state space and the amount of information available to the agents, whereas SL[1G] performance depends on the size and shape of the formula. These correlations are in line with the theoretical complexities of our model checking algorithms.

We can see that all the differences listed above stem from the different semantics used by the two fragments (item 2).

7.4 Summary

In this chapter, we evaluated both the theoretical and implementational contributions of our project. We first summarised our contributions in both areas and then discussed their strengths and weaknesses. The main strength of our solution is its novelty both on the theoretical and practical side as we have designed and implemented the first practical model checking algorithms for fragments of SL. The main weakness, which is low performance, can be overcome to a large extent by using various optimisation techniques. The impact of separate determinisation on SL[1G] model checking performance shows that the development of such techniques is one of the most promising future directions of this project (see Section 8.2).

We also provided a comprehensive comparison of SLK and SL[1G], both of which played a central role in our project. We found that all of the differences between the two fragments, the corresponding algorithms, and their implementations stem from their different semantics (SLK has imperfect recall with incomplete information, whereas SL[1G] has perfect recall with complete information).

Chapter 8

Conclusions

We tackled the problem of verifying *Strategy Logic* (SL) specifications. To do this, we designed the first practical model checking algorithms for two fragments of SL and implemented them as extensions of the existing MCMAS model checker. Our experimental results demonstrate that SL specifications, which can express various game-theoretic concepts including Nash equilibria, can be used in practice to verify and synthesise agents' behaviour. We have thereby made the first step towards bridging the gap between game-theory and practical model checking.

8.1 Summary of Work

To conclude the report, we summarise the work we have done and comment on whether we have met our objectives. The main goal of our project was to develop a model checker for SL. In order to achieve this goal, we first had to identify fragments of SL that are suitable for model checking. We found that the main obstacles to our goal were the undecidability of perfect recall semantics under incomplete information and the high model checking complexity of full SL. We addressed these obstacles by identifying two fragments of SL with desirable properties.

We first put forward a novel fragment of SL called *Epistemic Strategy Logic* (SLK), which is defined on imperfect recall semantics with incomplete information and adds support for epistemic modalities expressing agents' knowledge. We discussed its increased expressiveness due to the combination of strategic, temporal, and epistemic operators as well as its limitations caused by memoryless strategies. We proved that the model checking problem for SLK is PSPACE in both the size of the model and the formula. Finally, we designed an exponential-time model checking algorithm for SLK which we showed is correct, can be used to synthesise witness strategies, and admits an efficient symbolic implementation.

We then considered an existing fragment of SL called *One-Goal Strategy Logic* (SL[1G]), which is defined on perfect recall semantics with complete information. Although it is strictly more expressive than all logics in the ATL* hierarchy, it the same model checking complexity. We designed the first practical model checking algorithm for SL[1G] by reducing the problem to solving a two-player parity game. We proved that procedure is correct and has optimal worst-case time complexity. Furthermore, we explained how it can be used for general strategy synthesis and provided an efficient symbolic implementation. We also provided an optimisation technique for the class of SL[1G] formulas containing conjunctions of LTL formulas, which often occur in practice.

We implemented our novel model checking algorithms for both SLK and SL[1G] as extensions for the existing model checker MCMAS. We have thus developed the first model checker for fragments of SL and thereby achieved our main goal. The functionality of the extended tool includes witness strategy synthesis for both fragments and general strategy synthesis for SL[1G]. We also implemented the optimisation technique for SL[1G] as well as an experimental parallel implementation of the SLK model checking algorithm. Our experimental results demonstrate that the use of the fragments for verification and strategy synthesis is feasible in practice despite their high model checking complexities. Given the novelty of our solution and the promising results, we deem our project successful.

8.2 Future Work

Although we have achieved all our objectives outlined in Section 1.1 and implemented the first model checker for (a subset of) SL, the journey to practical SL model checking is nowhere near its end. The main obstacle, which needs to be overcome, remains the low performance of both fragments on large models. Possible future directions of this project include:

- **Automaton minimisation.** Both the standard translation (see Subsection 2.4.3) and the determinisation procedure (see Subsection 2.4.4), which we use in the SL[1G] model checking algorithm (see Figure 5.3), generate automata which are not necessarily minimal. One way to improve SL[1G] model checking performance is therefore to always minimise intermediate automata before performing the next step of the algorithm. A wide range of techniques for minimising both non-deterministic and deterministic automata which can be implemented symbolically have been presented in [77].

- **Determinisation of specific formula classes.** An alternative approach for reducing the size of intermediate automata generated by the SL[1G] model checking algorithm is to use different determinisation procedures (such as subset and breakpoint construction) for specific classes of LTL formulas (e.g. safety conditions). The feasibility of this approach has been demonstrated in [77].

- **Better parity game algorithm.** Our SL[1G] model checking procedure currently uses a very simple exponential-time algorithm for solving parity games. A wide variety of other more efficient algorithms, which we could use instead, are discussed in [79]. We expect that such a modification would have a significant impact on SL[1G] model checking performance since parity game solving is one of the main bottlenecks of the procedure (see Subsection 6.4.5). Likewise, more efficient algorithms for solving generalised parity games, which we use in our optimised SL[1G] model checking algorithm (see Subsection 5.2.5), have been proposed [28].

- **Alternative verification techniques.** In this project we focused purely on symbolic model checking using BDDs. However, there are many other verification techniques (see Table 2.2), some of which can handle much larger state spaces than BDDs. It is possible that a more efficient model checker for SLK, SL[1G], or a different SL fragment could be based on one of them. In our opinion, the most likely candidate is *bounded model checking*, which is used by some existing tools (e.g. NuSMV, see Subsection 2.5.4) for LTL model checking.

- **Alternative BDD packages.** As we have explained in Subsection 6.2.3, the main obstacle to improving SLK model checking performance through parallelism is the fact that the CUDD BDD package used by MCMAS has no built-in support for concurrency. There are two possible solutions to this problem: We could either modify the package, or replace it with a different one that supports parallelism (e.g. the BDDNOW package [71]).

- **Other SL fragments.** The next step in terms of augmenting the functionality of our extensions is to add support for other more expressive fragments of SL (see Section 3.2). The most likely candidate is the Disjunctive/Conjunctive-Goal Strategy Logic (SL[DG/CG]), which admits behavioural strategies and still has 2EXPTIME-COMPLETE model checking complexity despite being strictly more expressive than SL[1G]. Given the similar properties of the two fragments, we hope that support for SL[DG/CG] could be added by modifying our SL[1G] model checking algorithm. On the other hand, the high model checking complexity and non-behavioural semantics of the other SL fragments (SL[BG] and SL[NG]) might turn out to be a too big obstacle to practical model checking. Even if that is not the case, we expect the corresponding model checking algorithms to be very different from the one we proposed here for SL[1G].

- **Incomplete information.** While perfect recall semantics with incomplete information is undecidable in general (see Chapter 3), there exist special settings in which the problem becomes decidable. One example is the class of *two-player parity games with imperfect information*, in which the first player has incomplete knowledge of the game, whereas the second player has complete knowledge. An algorithm for solving such games together with a symbolic implementation was proposed in [21]. Recall that our model checking algorithm for SL[1G] constructs a parity game where the first player represents the existentially quantified strategy variables (see Subsection 5.2.1). Hence, we could use alternative SL[1G] semantics where the agents bound to the existentially quantified strategy variables in a formula have (the same) incomplete information and verify the formula by reducing the problem to solving a parity game with imperfect recall.

- **Improve support for subsumed logics.** Although SL[1G] subsumes all logics in the ATL* hierarchy, our SL[1G] extension does not have direct support for them, i.e. an ATL* formula must be translated (by the user) to an equivalent SL[1G] formula before it can be verified. Extending ISPL syntax (see Subsection 6.1.2) with ATL* formulas should be rather easy and it would greatly enhance the usability of the extension. Moreover, given that we have already implemented the procedures for translating LTL formulas to equivalent non-deterministic Büchi automata, it should not be too difficult to add support for efficient LTL and CTL* model checking as well.

Bibliography

[1] The BDD library. `http://www.cs.cmu.edu/~modelcheck/bdd.html`. Accessed 26/01/2014.

[2] Bison – GNU parser generator. `http://www.gnu.org/software/bison/`. Accessed 02/06/2014.

[3] Boolean satisfiability research group at Princeton. `http://www.princeton.edu/~chaff/software.html`. Accessed 26/01/2014.

[4] flex: The fast lexical analyzer. `http://flex.sourceforge.net/`. Accessed 02/06/2014.

[5] GNU general public license. `http://www.gnu.org/licenses/gpl.html`. Accessed 25/01/2014.

[6] GNU lesser general public license. `http://www.gnu.org/licenses/lgpl.html`. Accessed 24/01/2014.

[7] MCK. `http://cgi.cse.unsw.edu.au/~mck/pmck/`. Accessed 26/01/2014.

[8] MCMAS home page. `http://vas.doc.ic.ac.uk/software/mcmas/`. Accessed 25/01/2014.

[9] MCMAS v1.2.1: User manual. `http://vas.doc.ic.ac.uk/software/mcmas/documentation/`. Accessed 02/06/2014.

[10] NuSMV home page. `http://nusmv.fbk.eu/`. Accessed 24/01/2014.

[11] PRISM – probabilistic symbolic model checker. `http://www.prismmodelchecker.org/`. Accessed 26/01/2014.

[12] RSat homepage. `http://reasoning.cs.ucla.edu/rsat/`. Accessed 26/01/2014.

[13] VAS – verification of autonomous systems. `http://vas.doc.ic.ac.uk/`. Accessed 02/06/2014.

[14] MCK 1.0.0 – user manual. `http://cgi.cse.unsw.edu.au/~mck/pmck/mcks/docDownload/manual`, Feb. 2012. Accessed 26/01/2014.

[15] Accellera. Property specication language - reference manual - version 1.1. `http://www.eda.org/vfv/docs/PSL-v1.1.pdf`. Accessed 24/01/2014.

[16] R. Alur, L. de Alfaro, T. A. Henzinger, S. C. Krishnan, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Taşıran. Mocha user manual. `http://mtc.epfl.ch/software-tools/mocha/doc/`. Accessed 15/06/2014.

[17] R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. Mocha: Modularity in model checking. In A. J. Hu and M. Y. Vardi, editors, *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 521–525. Springer Berlin Heidelberg, 1998.

[18] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, Sept. 2002.

[19] S. Anthony. Amazon unveils 30-minute Prime Air quadcopter delivery service, but it's completely impractical. *ExtremeTech*, Dec. 2013. Accessed 25/01/2014.

[20] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In R. Alur and T. Henzinger, editors, *Proc. 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *LNCS*, pages 269–276. Springer, 1996.

[21] D. Berwanger, K. Chatterjee, M. De Wulf, L. Doyen, and T. A. Henzinger. Alpaga: A tool for solving parity games with imperfect information. *CoRR*, abs/0901.4728, 2009.

[22] A. Bianco and L. de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. Thiagarajan, editor, *Proc. 15th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'95)*, volume 1026 of *LNCS*, pages 499–513. Springer, 1995.

[23] N. Bulling, J. Dix, and W. Jamroga. Model checking logics of strategic ability: Complexity. In M. Dastani, K. V. Hindriks, and J.-J. C. Meyer, editors, *Specification and Verification of Multiagent Systems*, pages 125–159. Springer US, 2010.

[24] R. Carroll. CES 2014: driverless cars are coming and they want to be your friends. *The Guardian*, Jan. 2014. Accessed 25/01/2014.

[25] O. Carton and M. Michel. Unambiguous Büchi automata. *Theoretical Computer Science*, 297(1–3):37–81, 2003. Latin American Theoretical Informatics.

[26] P. Čermák, A. Lomuscio, F. Mogavero, and A. Murano. Few examples of interpreted systems. Personal communication, 2014.

[27] P. Čermák, A. Lomuscio, F. Mogavero, and A. Murano. MCMAS-SLK: A model checker for the verification of strategy logic specifications. *CoRR*, abs/1402.2948, 2014.

[28] K. Chatterjee, T. Henzinger, and N. Piterman. Generalized parity games. Technical Report UCB/EECS-2006-144, University of California, Berkeley, Nov. 2006.

[29] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of Cryptology*, 1:65–75, 1988.

[30] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV version 2: An opensource tool for symbolic model checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[31] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[32] A. Cimatti, M. Pistore, M. Roveri, and R. Sebastiani. Improving the encoding of LTL model checking into SAT. In *Proc. workshop on Verification Model Checking and Abstract InterpretationVMCAI'02*, volume 2294 of *LNCS*, Venice, Italy, Jan. 2002. Springer.

[33] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982.

[34] E. M. Clarke, O. Grumberg, and K. Hamaguchi. Another look at LTL model checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[35] C. Dima and F. L. Tiplea. Model-checking ATL under imperfect information and perfect recall semantics is undecidable. *CoRR*, abs/1102.4225, 2011.

[36] N. Eén and N. Sörensson. MiniSat page. `http://www.minisat.se/`. Accessed 26/01/2014.

[37] E. A. Emerson and J. Y. Halpern. "sometimes" and "not never" revisited: On branching versus linear time temporal logic. *J. ACM*, 33(1):151–178, Jan. 1986.

[38] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In C. Courcoubetis, editor, *Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 385–396. Springer Berlin Heidelberg, 1993.

[39] E. A. Emerson and C.-L. Lei. Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[40] E. A. Emerson, A. K. Mok, A. P. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 136–145. Springer Berlin Heidelberg, 1991.

[41] S. Even and A. Paz. A note on cake cutting. *Discrete Applied Mathematics*, 7(3):285–296, 1984.

[42] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*, volume 1 of *MIT Press Books*. The MIT Press, 2003.

[43] B. Farwer. ω-automata. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 3–21. Springer Berlin Heidelberg, 2002.

[44] O. Friedmann and M. Lange. Solving parity games in practice. In Z. Liu and A. Ravn, editors, *Automated Technology for Verification and Analysis*, volume 5799 of *Lecture Notes in Computer Science*, pages 182–196. Springer Berlin Heidelberg, 2009.

[45] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[46] E. Grädel, W. Thomas, and T. Wilke, editors. *Automata Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2002.

[47] M. Hilbert and P. López. The worlds technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[48] I. Hodkinson. C499: Modal and temporal logic. University Lecture, 2013. Department of Computing, Imperial College London.

[49] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, New York, NY, USA, 2004.

[50] M. Jurdziński. Deciding the winner in parity games is in UP $\cap$ co-UP. *Information Processing Letters*, 68(3):119–124, 1998.

[51] M. Jurdziński. Small progress measures for solving parity games. In H. Reichel and S. Tison, editors, *STACS 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 290–301. Springer Berlin Heidelberg, 2000.

[52] M. Kacprzak, W. Nabiałek, A. Niewiadomski, W. Penczek, A. Półrola, M. Szreter, B. Woźna, and A. Zbrzezny. VerICS 2007 – a model checker for knowledge and real-time. *Fundamenta Informaticae*, 85(1–4):313–328, 2008.

[53] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing testing with formal verification in Intel$^{®}$ Core^{TM} i7 processor execution engine validation. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 414–429. Springer Berlin Heidelberg, 2009.

[54] H. Klauck. Algorithms for parity games. In *Automata, Logics, and Infinite Games*, pages 107–129, 2001.

[55] S. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[56] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(2):128–142, 2004.

[57] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[58] F. Laroussinie, N. Markey, and G. Oreiby. On the expressiveness and complexity of ATL. In H. Seidl, editor, *Foundations of Software Science and Computational Structures*, volume 4423 of *Lecture Notes in Computer Science*, pages 243–257. Springer Berlin Heidelberg, 2007.

[59] N. G. Leveson and C. S. Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[60] J. Lind-Nielsen. BuDDy – a binary decision diagram package. `http://vlsicad.eecs.umich.edu/BK/Slots/cache/www.itu.dk/research/buddy/`. Accessed 26/01/2014.

[61] A. Lomuscio. *Knowledge Sharing among Ideal Agents.* PhD thesis, School of Computer Science, University of Birmingham, Birmingham, UK, June 1999.

[62] A. Lomuscio. C303: Systems verification. University Lecture, 2013. Department of Computing, Imperial College London.

[63] A. Lomuscio, H. Qu, and F. Raimondi. MCMAS: A model checker for the verification of multi-agent systems. In *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 682–688. Springer Berlin Heidelberg, 2009.

[64] A. Lomuscio and F. Raimondi. The complexity of model checking concurrent programs against CTLK specifications. In M. Baldoni and U. Endriss, editors, *Declarative Agent Languages and Technologies IV*, volume 4327 of *Lecture Notes in Computer Science*, pages 29–42. Springer Berlin Heidelberg, 2006.

[65] A. Lomuscio and F. Raimondi. Model checking knowledge, strategies, and games in multi-agent systems. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, AAMAS '06, pages 161–168, New York, NY, USA, 2006.

[66] A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[67] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by $\forall$-automata. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification*, volume 398 of *Lecture Notes in Computer Science*, pages 124–164. Springer Berlin Heidelberg, 1989.

[68] R. Mazala. Infinite games. In E. Grädel, W. Thomas, and T. Wilke, editors, *Automata Logics, and Infinite Games*, volume 2500 of *Lecture Notes in Computer Science*, pages 23–38. Springer Berlin Heidelberg, 2002.

[69] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[70] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[71] K. Milvang-Jensen, , and A. J. Hu. BDDNOW: A parallel BDD package. In *Proc. FMCAD*, pages 501–507. Springer, 1998.

[72] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. Reasoning about strategies: On the model-checking problem. *CoRR*, abs/1112.6275, 2011.

[73] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. A decidable fragment of strategy logic. *CoRR*, abs/1202.1309, 2012.

[74] F. Mogavero, A. Murano, G. Perelli, and M. Y. Vardi. What makes ATL* decidable? a decidable fragment of strategy logic. In M. Koutny and I. Ulidowski, editors, *CONCUR 2012 Concurrency Theory*, volume 7454 of *Lecture Notes in Computer Science*, pages 193–208. Springer Berlin Heidelberg, 2012.

[75] F. Mogavero, A. Murano, and L. Sauro. On the boundary of behavioral strategies. In *Logic in Computer Science (LICS), 2013 28th Annual IEEE/ACM Symposium on*, volume 0, pages 263–272, June 2013.

[76] F. Mogavero, A. Murano, and M. Y. Vardi. Reasoning about strategies. In K. Lodaya and M. Mahajan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, volume 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 133–144, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[77] A. Morgenstern. *Symbolic controller synthesis for LTL specifications.* PhD thesis, 2010.

[78] A. Morgenstern and K. Schneider. From LTL to symbolically represented deterministic automata. In F. Logozzo, D. A. Peled, and L. D. Zuck, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 4905 of *Lecture Notes in Computer Science*, pages 279–293. Springer Berlin Heidelberg, 2008.

[79] J. Obdržálek. *Algorithmic Analysis of Parity Games.* PhD thesis, University of Edinburgh, 2006. Submitted: January 31, 2006. Examined: May 29, 2006.

[80] A. Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, Oct 1977.

[81] F. Raimondi and A. Lomuscio. Automatic verification of multi-agent systems by model checking via OBDDs. *Journal of Applied Logic*, 2005.

[82] S. Safra. On the complexity of ω-automata. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 319–327, Oct 1988.

[83] K. Schneider. *Verification of Reactive Systems.* Texts in Theoretical Computer Science. Springer Berlin Heidelberg, 2004.

[84] P. Schnoebelen. The complexity of temporal logic model checking. *Advances in Modal Logic*, 4:393–436, 2002.

[85] F. Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`. Accessed 26/01/2014.

[86] M. Y. Vardi and P. Wolper. Reasoning about infinite computations. *Inf. Comput.*, 115(1):1–37, 1994.

[87] B. Woźna and A. Zbrzezny. Bounded model checking for the universal fragment of CTL, 2002.

[88] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1–2):135–183, 1998.

[89] M. Zimmermann. Optimal Bounds in Parametric LTL Games. *ArXiv e-prints*, June 2011.

Appendix A

Toy Model ISPL Code

We provide here an ISPL description of the toy model introduced in Section 3.3.

```
Agent Environment
  Obsvars:
    state : {game, p1win, p2win};
  end Obsvars
  Actions = {idle};
  Protocol:
    Other : {idle};
  end Protocol
  Evolution:
    state = p1win if state = game and
            (Player1.Action = rock and Player2.Action = scissors or
             Player1.Action = paper and Player2.Action = rock or
             Player1.Action = scissors and Player2.Action = paper);
    state = p2win if state = game and
            (Player2.Action = rock and Player1.Action = scissors or
             Player2.Action = paper and Player1.Action = rock or
             Player2.Action = scissors and Player1.Action = paper);
  end Evolution
end Agent

Agent Player1
  Vars:
  end Vars
  Actions = {rock, paper, scissors, idle};
  Protocol:
    Environment.state = game: {rock, paper, scissors};
    Other: {idle};
  end Protocol
  Evolution:
  end Evolution
end Agent

Agent Player2
  Vars:
  end Vars
  Actions = {rock, paper, scissors, idle};
  Protocol:
    Environment.state = game: {rock, paper, scissors};
    Other: {idle};
  end Protocol
  Evolution:
  end Evolution
end Agent

Evaluation
  p1 if Environment.state = p1win;
  p2 if Environment.state = p2win;
end Evaluation

InitStates
  Environment.state = game;
end InitStates

Groups
  g = {Player1, Player2};
end Groups

Formulae
  -- Existing Tool (examples/toy_model_existing.ispl)
  EF p1;
  <g>G !(p1 or p2);

  -- SLK Extension (examples/toy_model_slk.ispl)
  <<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y) G !(p1 or p2);

  -- SL[1G] Extension (examples/toy_model_sl1g.ispl, examples/toy_model_sl1g2.ispl)
  #PR <<e>> (Environment, e) <<x>> (Player1, x) <<y>> (Player2, y) G !(p1 or p2);
  #PR [[e]] (Environment, e) [[x]] (Player1, x) <<y>> (Player2, y) G !(p1 or p2);
end Formulae
```