BEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

# An Automata-Theoretic Approach for the Verification of Strategic Abilities in Multi-Agent Systems

*Supervisor:*
Dr. Francesco Belardinelli

*Author:*
Sofía García de Blas García-Alcaide

*Second Marker:*
Dr. Matthew Wicker

June 12, 2025

Submitted in partial fulfillment of the requirements for the Bachelor's in Computing of Imperial College London

**Abstract**

Verifying multi-agent systems is essential to ensure their correctness and safety. Formal verification techniques such as model checking are effective tools for this purpose, especially when combined with logics like ATL* that can express strategic reasoning among agents. While multi-agent systems are often modelled using infinite traces to capture ongoing interactions, many scenarios naturally lend themselves to finite-trace modelling where executions terminate after a finite number of steps. In such cases, model checking can avoid the complexities of $\omega$-automata and instead use finite automata, yielding simpler algorithms.

This project proposes a novel symbolic algorithm for finite-trace verification of ATL$^*$, and implements this along with the explicit-state algorithm introduced by Belardinelli et al. For infinite traces, we design and implement two hybrid algorithms for ATL$^*$ model checking over infinite traces: one based on the classical nondeterministic Rabin tree automata (NRTA) construction, and another by a novel symbolic reduction to parity games. We evaluate these algorithms using a synthetic benchmark. Our results demonstrate that the symbolic approach significantly outperforms the explicit-state representation, highlighting the advantages of symbolic techniques in mitigating the state explosion problem. Furthermore, we find that the NRTA-based approach incurs substantial computational overhead, making it impractical for large systems. In contrast, our parity-game-based algorithm offers a more scalable and efficient solution for infinite-trace verification, also outperforming MCMAS' SL[1G] model checker.

These results confirm that finite-trace model checking yields substantial performance benefits over infinite-trace verification, and provide a comprehensive toolset for verifying multi-agent systems with ATL$^*$.

## Acknowledgements

First, I would like to express my deepest gratitude to my supervisor, Dr. Francesco Belardinelli, for his continuous support throughout this project, for the weekly meetings, and for his invaluable advice. I would also like to thank my second marker, Dr. Matthew Wicker, for his helpful feedback after the interim report and for his availability to discuss the project throughout the year.

To my friends: thank you for making even the most stressful parts of this project so much more fun, and especially to Siddhant Chauhan for actually showing up to our 8am meetings to work on the project and for listening to me practice before every meeting.

I am especially grateful to my family - my parents, my sister, and my partner - for their unwavering love and support throughout these three years. To my dad, thank you for reading this report more times than I can count and for the endless conversations we had about the project. To my mum, thank you for always believing in me more than I believe in myself. And to Jaume, thank you for being there during my breakdowns and moments of doubt, and for always supporting me. I am incredibly fortunate to have you all in my life.

# Contents

# Chapter 1

# Introduction

## 1.1 Problem Statement

Ensuring the safety and reliability of artificial intelligence (AI) systems is a critical challenge, particularly as AI becomes increasingly integrated into high-stakes applications. While traditional approaches - such as peer reviews and automatic testing - remain the standard techniques for software verification ([4], [9]), their limitations have become more evident in the context of advanced AI frameworks, like large language models (LLMs). Despite extensive pre-deployment testing, LLMs can still produce unsafe content that bypasses their safety mechanisms through sophisticated prompt engineering techniques, such as "jailbreak" attacks [2]. This highlights a critical gap: while empirical testing may identify known vulnerabilities, it often fails to anticipate novel adversarial techniques. As AI systems become more advanced and are deployed in dynamic environments, formal methods of verification have gained increasing traction in software verification, due to the mathematically rigorous methods they offer to ensure safety and reliability of the systems.

## 1.2 Proposed Solution/Methodology

Formal verification techniques provide a rigorous framework for ensuring system reliability. Among these, **model checking** has emerged as a robust approach, enabling the exhaustive exploration of all possible system states to verify the validity of specified properties [4]. At the heart of formal verification are three essential components as defined by Mcmillan [56]: a formal model of the system, a specification language to express desired properties, and an algorithmic methodology for verifying these properties.

Central to model checking is the use of **temporal logics** as specification languages, such as Linear Temporal Logic (LTL), Computation Tree Logic (CTL), or Alternating-Time Temporal Logic (ATL). Each variant offers unique capabilities for reasoning about system behaviours over time, with ATL incorporating strategic reasoning, enabling the verification of what components of a system can individually or collectively achieve under specific strategies - a key focus of this work.

For decades, formal verification techniques - such as model checking - have been instrumental in verifying concurrent systems like cache coherency protocols, which face challenges similar to those of AI systems, including complex interactions and nondeterministic behaviour ([4]; [44]). Building on their success, these techniques offer a rigorous approach to ensuring safety in AI environments, particularly for **multi-agent systems** (MAS).

MAS consist of autonomous agents that interact and collaborate to achieve shared or individual goals [30]. Their decentralised decision-making and strategic interactions make them ideal candidates for formal verification, using logics like ATL. To give a real example, Nam and Kil [59] demonstrated how ATL model checking can verify the correctness of blockchain smart contracts, which are inherently multi-agent systems. ATL enabled the detection of vulnerabilities, such as the DAO (Decentralised Autonomous Organisation) attack, which exploited a re-entrancy flaw in the contract code. This vulnerability allowed recursive withdrawals, leading to the theft of approximately $50 million worth of Ether [68]. This underscores once again the critical role of temporal logics in ensuring correctness in systems with strategic interactions.

In addition to a specification language, model checkers require a formal model of the system to be verified. Traditionally, system behaviours in model checking have been modelled as infinite interactions with the environment, reflecting the nature of reactive systems that continuously respond to inputs over an unbounded timeline [41]. While this accurately represents domains such as operating systems or network protocols where ongoing behaviour is critical, many other domains naturally exhibit finite behaviours, with executions that terminate upon achieving a specific outcome.

Adopting finite executions for modelling these systems not only provides a more intuitive representation, but also addresses significant computational challenges associated with infinite traces. For instance, model checking properties over infinite traces requires Büchi automata complementation, which has a lower bound blowup of n! [71]. In contrast, modelling over finite traces allows the use of automata over finite words, significantly simplifying the algorithms involved in the model checking procedure [7].

**Temporal logic over finite traces** provides a powerful framework for expressing properties in planning problems, where the goal is to generate sequences of actions that transition a system from an initial state to a desired goal state [35]. Camacho et al. demonstrated the advantages of using Linear Temporal Logic over finite traces ($\text{LTL}_f$) in nondeterministic planning tasks, such as robot navigation or task scheduling, offering a concise and computationally efficient way to handle finite episodes [15]. In business process management, temporal logics like $\text{LTL}_f$ facilitate the specification and monitoring of compliance with high-level meta-constraints, such as regulatory requirements or operational deadlines [37]. Similarly, in automated negotiation protocols, which require agents to interact and reach agreements within finite interactions [48], temporal logic over finite traces provides a formal framework for specifying rules, for instance ensuring that counteroffers are made within a bounded number of steps. These applications illustrate the versatility and computational efficiency of temporal logics over finite traces in addressing temporally extended behaviours across a range of domains characterised by finite executions.

Building on these insights, the goal of this project is to define and develop efficient model checking algorithms for ATL$^*$, a more expressive variant of ATL, over both finite and infinite traces, recognising that many real-world systems exhibit behaviours best captured by either semantics, depending on the application domain. By supporting both semantics, the project aims to provide a unified framework for the verification of strategic reasoning under different execution models and to assess their practical differences in terms of computational performance.

## 1.3 Contributions

This project aims to advance the state of the art in formal verification for multi-agent systems by developing, adapting and evaluating model checking algorithms for both finite- and infinite-trace variants of ATL*. The core contributions of this work are:

1. *Symbolic Algorithm and implementation for ATL$^*_f$ model checking*: We present a novel symbolic model checking algorithm for the ATL$^*_f$ logic in Chapter 3, building on the theoretical foundations proposed in [7]. The symbolic formulation developed here includes several design choices that differ substantially from the theoretical presentation and enable scalable implementation. This is followed by a complete implementation of both the explicit and symbolic versions, comprising the first working tools for verifying ATL$^*$ over finite traces.

2. *Hybrid symbolic-explicit model checking algorithms for ATL$^*$ over infinite traces:* Two new algorithmic approaches for verifying ATL$^*$ over infinite traces are presented in Chapter 4. The first is based on a hybrid construction of Rabin tree automata and a symbolic CGS execution tree. The second develops a novel ATL$^*$-specific symbolic reduction to parity games.

3. *Modelling of a strategic cybersecurity scenario:* In Chapter 6, a concrete attacker-defenders multi-agent system is modelled to demonstrate the applicability of the symbolic ATL$^*_f$ model checker to realistic verification tasks.

4. *Comparative analysis of finite vs. infinite-trace verification:* The project empirically investigates the practical performance trade-offs between finite- and infinite-trace ATL* model checking, as well as between different algorithms in both domains. The results of these evaluations are presented in Chapters 5 and 7.

These algorithms are integrated into the final product of this project: a C++ model checking tool that supports both finite and infinite-trace verification of ATL$^*$. Users can provide a system specification and select among the implemented algorithms to verify the property. This tool encapsulates the project's theoretical and practical contributions, offering a flexible and scalable solution for verifying strategic properties in multi-agent systems.

# Chapter 2

# Background

To provide the necessary context for understanding the development and application of the ATL* model checking algorithms, this background section introduces the foundational concepts underpinning the methodology. We begin with the introduction of alternating-time temporal logic (ATL) with its syntax and semantics over concurrent game structures (CGSs), followed by an overview of automata theory as a tool for representing system models and specifications expressed in temporal logic. This leads into a general discussion of model checking, including its challenges - most notably the state explosion problem - and strategies for their mitigation. Finally, we give an overview of state-of-the-art model checkers which highlights the current landscape and provide the complexity results for the model checking problem for several relevant logics.

## 2.1 Alternating-Time Temporal Logic

The origins of temporal logics can be traced back to philosophical inquiries into the nature of time and its relationship with truth. Aristotle's famous "sea battle" paradox, which questions whether statements about future events (e.g. "There will be a sea battle tomorrow") are currently true or false, inspired centuries of debate on the interplay between time and determinism. In the mid-20th century, Arthur Prior formalised these ideas through tense logic, a precursor to modern temporal logics, emphasising the relationship between propositions and their temporal contexts [38], [12]. Initially developed to reason about natural language, temporal logics were later adapted for computational systems, providing a powerful framework for specifying and verifying system behaviours. Today, they play a pivotal role in system verification, enabling rigorous analysis of critical properties such as safety ("nothing bad ever happens") and liveness ("something good eventually happens") across domains ranging from hardware design to distributed systems and multi-agent interactions [4].

Linear Temporal Logic (LTL), Computation Tree Logic (CTL), and Alternating-Time Temporal Logic (ATL) are three widely used variants. While LTL focuses on reasoning along linear sequences of events, making it well-suited for properties like "event B must eventually follow event A", CTL extends this by reasoning over branching time structures, allowing properties to be expressed across multiple possible futures.

Alternating-Time Temporal Logic (ATL), introduced by Alur, Henzinger, and Kupferman, extends traditional temporal logics like LTL and CTL by enabling reasoning about the strategic abilities of agents or coalitions in open systems. While CTL allows existential ($\exists$) and universal ($\forall$) quantifica-

tion over possible paths in a computation tree, ATL introduces strategic quantifiers ($\langle\langle A \rangle\rangle$) to express whether a coalition of agents has a strategy to achieve a desired outcome, regardless of the actions of other agents. ATL*, a more expressive extension of ATL, allows the nesting of both strategic and temporal operators, so it can express more complex properties than ATL [1]. Interpreted over **Concurrent Game Structures (CGS)**, ATL* is particularly powerful for modelling and verifying multi-agent systems, where outcomes are determined by the interactions of multiple agents.

### 2.1.1 ATL and ATL* Syntax

We now state the syntax for the language ATL* and its fragment ATL as introduced in [1]. Let $P$ be the set of atomic propositions and $Ag$ the set of agents. ATL* formulas are the set of state ($\varphi$) formulas, which are built using path formulas ($\psi$) with Boolean operators and strategic quantifiers.

$$\varphi ::= \texttt{true} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \langle\langle A \rangle\rangle \psi$$

$$\psi ::= \varphi \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \mathsf{X}\psi \mid \psi_1 \mathsf{U} \psi_2$$

where $p \in P$ and $A \subseteq Ag$.

The temporal operators X (next) and U (until) are interpreted as follows:

- $\mathsf{X}\varphi$: $\varphi$ holds in the next state.
- $\varphi_1 \,\mathsf{U}\, \varphi_2$: $\varphi_2$ holds at some future state, and $\varphi_1$ holds at all states up to that point.

Other propositional operators, such as false, $\vee$ (or), $\rightarrow$ (implies), and $\leftrightarrow$ (double implication), can be derived in the conventional manner from the core operators `true`, $\wedge$, and $\neg$. Additional temporal operators are also derivable from X and U:

- $\mathsf{F}\varphi$ (finally): $\varphi$ holds at some point in the future. $\mathsf{F}\varphi \equiv \texttt{true}\mathsf{U}\varphi$
- $\mathsf{G}\varphi$ (globally): $\varphi$ holds now and at every future moment. $\mathsf{G}\varphi \equiv \neg\mathsf{F}\neg\varphi$

For simplicity, this report includes these derived operators in the syntax.

The strategic quantifier $\langle\langle A \rangle\rangle \psi$ ("the agents in A can enforce"), means that the agents in A can collaborate with each other to achieve a joint strategy such that, no matter what the other agents do, the property $\psi$ will hold. We introduce its dual strategic quantifier $[\![A]\!]\psi ::= \neg\langle\langle A \rangle\rangle\neg\psi$ ("no matter what the agents in A do"), meaning that the agents in A cannot cooperate to avoid $\psi$.

The syntax of the ATL* fragment ATL changes the path formulas to avoid nesting Boolean connected with strategic quantifiers. ATL is strictly less expressive than ATL*. Consider the following property, "The defenders ($D_1$, $D_2$) have a strategy to ensure that it is always possible for the system to eventually reach a safe state". This can be expressed in ATL* as

$$\langle\langle D_1, D_2 \rangle\rangle\ \mathsf{GF}Safe$$

However, it is impossible to express this property in ATL because the formula requires a nesting of the temporal operators which ATL does not support.

**Linear Temporal Logic**

Linear-Time Temporal Logic (LTL) was introduced by Amir Pnueli in [61] as a formalism for reasoning about the temporal behaviour of reactive systems. In the context of ATL, LTL can be seen as a special case where the system consists of a single agent, and reasoning is confined to linear paths rather than branching strategies. The syntax of LTL formulas, as defined in standard references [61], [4], is presented below.

$$\varphi ::= \texttt{true} \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \mathsf{X}\varphi \mid \varphi_1 \,\mathsf{U}\, \varphi_2$$

where $p \in P$ represents an atomic proposition from the set of atomic propositions $P$.

While LTL was originally developed over infinite traces for systems with unbounded executions, its semantics over finite traces was proposed in [26] to apply it in verification for systems with fixed execution lengths.

### 2.1.2 Concurrent Game Structures

Linear-Time Temporal Logic (LTL) and Computation Tree Logic (CTL) are typically interpreted over Kripke structures, which model the system being verified as a closed entity where all transitions are determined by the state of the system. However, this framework is insufficient for open systems, where the system interacts with the environment or other agents, and its behaviour is influenced by the choices of these external actors [1]. To address this limitation, Alur et al. introduced Concurrent Game Structures (CGS) as a generalisation of Kripke structures. CGS model the system as a multi-agent framework, where the next state is determined by the joint actions of all agents operating concurrently, rather than a predefined global transition, making them particularly suited for reasoning about strategic interactions in open systems.

We now give the formal definition of a CGS following [1].

**Definition of a CGS**

**Definition 1.** *A Concurrent Game Structure is a tuple $G = \langle Ag, P, Act_{a\in Ag}, S, s_0, \delta, \lambda\rangle$ where:*

- *$Ag$ is the finite non-empty set of agents*
- *$P$ is the finite non-empty set of atomic propositions*
- *$Act_a$ is the finite non-empty set of actions for each agent $a \in Ag$*
- *$S$ is the finite non-empty set of states, and $s_0 \in S$ is the initial state of the system*
- *$\delta : S \times Jact \to S$ is the transition function where the set of joint actions $Jact$ is defined as $Jact = \prod_{a \in Ag} Act_a$ where each member of $Jact$ is a tuple with one action per agent*
- *$\lambda : S \to 2^P$ is the labelling function that assigns to each state $s$ the set of propositional atoms that are true in $s$.*

To model check ATL$^*_f$ we extend this definition with a set of final states $F \subseteq S$ to identify terminating executions, as in [7].

Following [7], we define the following concepts related to CGS on both finite and infinite systems, but only considering perfect recall systems.

A **history** is a finite sequence of states $\pi = \pi(1)\pi(2)\ldots\pi(n)$, where each $\pi(i)$ satisfies the transition function $\delta$. Formally, for every $i \leq n$ there exists a joint action $J \in Jact$ such that: $\delta(\pi(i), J) = \pi(i+1)$. An **infinite run** is an infinite such sequence. In the finite-trace setting, we additionally define a **path** as a history ending in a final state $\pi(n) \in F$. The set of all histories and runs are defined as $Hist$ and $Run$ respectively.

A **strategy** for an agent $a$ is a function $\sigma_a : Hist \to Act_a$, mapping histories to actions. In infinite-trace settings, strategies are similarly defined over prefixes of runs. A strategy is considered to have **perfect recall** if the action selected depends on the entire history/path and not just on the current state. The set of all perfect recall strategies is denoted as $\sum_R(G)$.

A **joint strategy** $\sigma_A : A \to \sum_R(G)$ is a function that associates to a subset A of the agents of the system a set of strategies, one for each agent.

For a state $s \in S$, the **finite** (resp. **infinite**) outcome of a joint strategy $\sigma_A$ from $s$, denoted $\text{out}_f(s, \sigma_A)$ (resp. $\text{out}_\infty(s, \sigma_A)$), is the set of finite paths (resp. runs) $\pi$ consistent with $s$ and $\sigma_A$. A path (resp. run) is **consistent** if

- $\pi(1) = s$
- For every $i < length(\pi)$, there exists a joint action $J \in Jact$ such that $\pi(i+1) = \delta(\pi(i), J)$ and $J(a) = \sigma_A(a)(\pi_{\leq i})$ for each agent $a$.

#### Intrusion Detection Scenario as a CGS

To illustrate the concepts of CGSs, we model a simple intrusion detection system involving two agents: a defender (D) and an attacker (A). The attacker attempts to compromise the system, and the defender may scan or remain idle. The CGS $G = \langle Ag, P, \{Act_a\}, S, s_0, F, \delta, \lambda\rangle$ is defined as follows:

- $Ag = \{D, A\}$, $P = \{\texttt{Compromised}, \texttt{Detected}\}$
- $Act_D = \{\texttt{Scan}, \texttt{Idle}\}$, $Act_A = \{\texttt{Exploit}, \texttt{Wait}\}$
- States: $S = \{s_0, s_1, s_2, s_3\}$, with $s_0$ as initial and $F = \{s_2, s_3\}$
- Labelling: $\lambda(s_2) = \{\texttt{Compromised}\},\ \lambda(s_3) = \{\texttt{Detected}\},\ \lambda(s_0) = \lambda(s_1) = \emptyset$

The transition function $\delta$ is defined in Table 2.1:

| **State** | **Joint Action** $(Act_D, Act_A)$ | **Next State** |
|---|---|---|
| $s_0$ | (Idle, Exploit) | $s_1$ |
| $s_0$ | (Scan, Exploit) | $s_3$ |
| $s_0$ | (Idle, Wait) | $s_0$ |
| $s_1$ | (Idle, Wait) | $s_2$ |
| $s_1$ | (Scan, Wait) | $s_3$ |

**Table 2.1:** Transition function $\delta$ for the intrusion detection CGS.

Example histories and paths include: $\pi_1 = s_0 s_1 s_2$ representing a successful compromise, and $\pi_2 = s_0 s_3$ showing immediate detection.

### 2.1.3 ATL* Semantics

The syntax of ATL$^*_f$ is identical to that of ATL$^*$, but their semantics differ due to the distinction between finite and infinite traces. Following [7], we define a parameterised semantics using $x \in \{\infty, f\}$ to denote infinite and finite trace interpretations, respectively.

1. **State formulas** $\varphi$: $(G, s) \models_x \varphi$ denotes that $\varphi$ holds at state $s \in S$ in CGS $G$, which is extended with final states if $x = f$.
2. **Path formulas** $\psi$: $(G, \pi) \models_x \psi$ denotes that $\psi$ holds along a trace $\pi$, where $\pi \in$ *Run* if $x = \infty$, and $\pi \in$ *Path* if $x = f$.

Let $\pi_{\geq i}$ denote the suffix of $\pi$ starting at index $i$. The semantics are defined recursively as follows:

$$
\begin{aligned}
(G, s) \models_x p &\iff p \in \lambda(s) \\
(G, s) \models_x \neg\varphi &\iff (G, s) \not\models_x \varphi \\
(G, s) \models_x \varphi \wedge \varphi' &\iff (G, s) \models_x \varphi \text{ and } (G, s) \models_x \varphi' \\
(G, s) \models_x \langle\langle A \rangle\rangle \psi &\iff \text{there exists a joint strategy } \sigma_A \in \Sigma_R(G) \text{ such that} \\
&\qquad \text{for all } \pi \in \text{out}_x(s, \sigma_A),\ (G, \pi) \models_x \psi \\
(G, \pi) \models_x \varphi &\iff (G, \pi(0)) \models_x \varphi \\
(G, \pi) \models_x \neg\psi &\iff (G, \pi) \not\models_x \psi \\
(G, \pi) \models_x \psi_1 \wedge \psi_2 &\iff (G, \pi) \models_x \psi_1 \text{ and } (G, \pi) \models_x \psi_2 \\
(G, \pi) \models_x \mathsf{X}\psi &\iff |\pi| > 1 \text{ and } (G, \pi_{\geq 1}) \models_x \psi \\
(G, \pi) \models_x \psi_1 \,\mathsf{U}\, \psi_2 &\iff \text{there exists } j < |\pi| \text{ such that } (G, \pi_{\geq j}) \models_x \psi_2, \\
&\qquad \text{and for all } 0 \leq k < j,\ (G, \pi_{\geq k}) \models_x \psi_1
\end{aligned}
$$

We write $G \models_x \varphi$ as shorthand for $(G, s_0) \models_x \varphi$, where $s_0$ is the initial state of $G$.

### 2.1.4 ATL* over Finite Traces in Intrusion Detection

To illustrate the semantics of ATL$^*_f$, we revisit the intrusion detection scenario from Section 2.1.2. Consider the formula:

$$\langle\langle A \rangle\rangle \,\mathsf{F}\, \texttt{Compromised}$$

This states that the attacker $A$ has a strategy to ensure that the system is eventually compromised on all finite executions consistent with that strategy.

In the CGS, if the defender $D$ remains idle in $s_0$ and $A$ chooses to `Exploit`, the system reaches $s_1$. Then, if $A$ waits again and $D$ continues to idle, the system transitions to $s_2$, where `Compromised` holds. Thus, there exists a strategy for $A$ leading to compromise, and the formula is satisfied.

This example illustrates how ATL$^*_f$ captures strategic reasoning over finite executions, and how system outcomes depend on the interplay between agent strategies.

## 2.2 Automata Theory

Automata play a central role in model checking, offering a formal mechanism for representing system behaviours and verifying temporal logic properties. By capturing properties as languages over words, automata enable the translation of logic specifications into computational models. These techniques form the basis of many model checking algorithms for verifying both finite and infinite-state systems. The concepts introduced here follow the explanations in ([4], [46]).

### 2.2.1 Automata on Finite Words

**Definition 2.** *A nondeterministic finite automaton (NFA) is a tuple $A = \langle Q, \Sigma, \delta, Q_0, F \rangle$ where*

- *$Q$ is a finite set of states*
- *$\Sigma$ is an alphabet, i.e. a finite set of symbols*
- *$\delta : Q \times \Sigma \to 2^Q$ is a transition functions which determines from the current state and an input symbol what are the possible next states of the system*
- *$Q_0 \subseteq Q$ is a set of initial states*
- *$F \subseteq Q$ is a set of final states*

**Definition 3.** *Let $w$ be a finite word $w = A_1 \dots A_n \in \Sigma^*$ and $A$ an NFA as defined in Definition 2. A run for $w$ in $A$ is a finite sequence of states $q_0 q_1 \dots q_n$ such that*

- *$q_0 \in Q_0$ (the run starts in an initial state)*
- *$q_{i+1} \in \delta(q_i, A_i + 1)$ for all $0 \leq i < n$*

*A run $q$ is **accepting** if it ends in a final state, i.e. if $q_n \in F$. A finite word $w$ is accepting if any of the runs for $w$ in $A$ are accepting.*

*The **accepted language** of an NFA $A$, $\mathcal{L}(A)$ is the set of all finite words $w$ that are accepted by $A$.*

NFAs are nondeterministic since for every word there may be more than one possible run. In contrast, a deterministic finite automaton (DFA) has one unique run for each input word.

**Definition 4.** *A deterministic finite automaton is a tuple $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ where the symbols are defined in the same way as for an NFA but with the following changes*

- *$q_0$ is the single, unique initial state*
- *$|\delta(q, A)| \leq 1$ for all states $q$ and input symbols $A$*

A DFA is considered **total** if $|\delta(q, A)| = 1$ for all states $q$ and input symbols $A$.

NFAs and DFAs are equivalent in expressive power, meaning they both recognise the class of **regular languages**. We can obtain an equivalent DFA from an NFA through the **powerset construction** which has a worst-case complexity of $O(2^n \cdot |\Sigma|)$ [63].

Given a DFA, the process of **minimisation** produces an equivalent DFA with the minimal number of states. The minimisation algorithm operates in $O(NlogN)$ time for a DFA with $N$ states [45].

### 2.2.2 Automata on Infinite Words

While finite automata accept finite words and recognise the class of regular languages, certain systems require the representation of infinite behaviours, such as those in reactive systems. To address this, automata on infinite words, or $\omega$-automata, are used. The simplest variant is the nondeterministic Büchi automaton (NBA), which accepts the class of $\omega$-regular languages and forms the foundation for verifying properties of infinite-state systems.

**Definition 5.** *A nondeterministic Büchi automaton (NBA) is a tuple $A = \langle Q, \Sigma, \delta, Q_0, F \rangle$ with the same syntax and interpretation of the symbols as an NFA (see Definition 2).*

**Definition 6.** *A run for $\sigma = A_0, A_1, \ldots \in \Sigma^\omega$ denotes an infinite sequence of states in $Q$ such that $q_0 \in Q_0$ and consecutive states are related through the transition relation $\delta$. A run $\sigma$ is accepting if $q_i \in F$ for infinitely many indices $i \in \mathbb{N}$. This is known as the **Büchi acceptance condition**. The accepted language of an NBA, denoted $\mathcal{L}(A)$, is the set of all $\omega$-words $\sigma$ where there exists one accepting run for $\sigma$.*

It is worth noting that while Deterministic Büchi Automata exist, they have a lower expressive power than their nondeterministic counterparts. This is not the case with Deterministic Rabin or Parity Automata.

Büchi automata are essential in the automata-theoretic approach to model checking, particularly for verifying LTL formulas over infinite traces. The process as presented in [73] involves constructing a Büchi automaton that represents the behaviours satisfying the LTL formula, which is at most exponential in the size of the formula [4]. To obtain an automaton that represents the violating behaviours of the formula, the **Büchi Complement** is performed. This automaton is combined with the system model into a product automaton, whose emptiness is checked to determine whether the formula holds. If the product automaton's language is empty, the system satisfies the formula; otherwise a counter example exists. Efficient algorithms for the emptiness check are described in [24], [72].

We also have other types of acceptance conditions. Here we present the Rabin and Parity conditions, which are all equally expressive and recognize the class of $\omega$-regular languages.

A Rabin acceptance condition is a set $F = \{(G_1, B_1), ..., (G_n, B_n)\}$ where each $B_i, G_i \subseteq S$. A run $\sigma$ is accepting if

$$\exists i \in \mathbb{N}.1 \leq i \leq n \wedge (\text{Inf}(\sigma) \cap F_i = \emptyset) \wedge (\text{Inf}(\sigma) \cap G_i \neq \emptyset)$$

Parity acceptance is a special type of Rabin acceptance where the accepting pairs can be ordered with respect to set inclusion. A parity acceptance is a set $F = \{P_1, P_2, \ldots, P_n\}$ where each $P_i \subseteq S$, and a run is accepting if

$$\min_i \ P_i \cap \text{Inf}(\sigma) \neq \emptyset$$

is odd. This is considered a min-odd parity condition; other types exist such as max-odd or min-even. We will use these different types of acceptance conditions in the ATL$^*$ over infinite traces model checking algorithms, since their different structures give rise to different algorithms.

### 2.2.3 Alternating Automata

Alternating Automata generalise nondeterministic automata by allowing transitions to be expressed as positive Boolean formulas over successor states. This expressiveness enables both existential and

universal branching within the same model, making them especially suitable for symbolic automata constructions and LTL-to-automata translations. We follow the general framework in [18, 72] and define alternating automata over words in a way that encompasses both finite and infinite trace semantics.

**Definition 7.** *An **Alternating Automaton** over words is a tuple $A = \langle Q, \Sigma, \delta, q_0, \mathcal{F} \rangle$, where:*

- *$Q$ is a finite set of states,*
- *$\Sigma$ is the input alphabet,*
- *$q_0 \in Q$ is the initial state,*
- *$\delta : Q \times \Sigma \to \mathcal{B}^+(Q)$ is the transition function, mapping a state and input symbol to a positive Boolean formula over $Q$,*
- *$\mathcal{F}$ is an acceptance condition over infinite or finite branches*

**Runs.** A run of $A$ on a word $w = a_0 a_1 a_2 \ldots \in \Sigma^* \cup \Sigma^\omega$ is a rooted tree where:

- Each node at depth $i$ is labelled by a state in $Q$ and corresponds to the symbol $a_i$ in the input word.
- The children of a node labelled $q$ at depth $i$ are determined by the satisfaction of the Boolean formula $\delta(q, a_i)$. For instance, if $\delta(q, a_i) = q_1 \wedge (q_2 \vee q_3)$, the run must branch to satisfy this structure.

**Acceptance.** The acceptance condition $\mathcal{F}$ determines whether a run is accepting:

- For finite traces, $\mathcal{F} = F \subseteq Q$, and a run is accepting if **all leaves** are in $F$.
- For infinite traces, $\mathcal{F}$ may be a Büchi, Rabin, or parity condition. Each branch of the run must satisfy the condition individually.

A word is accepted if there exists a run tree rooted at $q_0$ where all branches satisfy the acceptance condition.

**Definition 8** ([58])**.** *An alternating automaton $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ is a **Weak Alternating Automaton (WAA)** if:*

- *The state set $Q$ is partitioned into disjoint subsets, such that each $Q_i$ is either:*
  - Accepting: $Q_i \subseteq F$, *or*
  - Rejecting: $Q_i \cap F = \emptyset$
- *There exists a partial order $<$ over the set of partitions $\{Q_i\}$ such that for any $q \in Q_k$ and $q' \in Q_j$ where $q'$ appears in$\delta(q, a)$, it holds that $Q_j \leq Q_k$*

*This means that every run of the automaton eventually remains in a single partition $Q_i$, and the run is accepting if and only if this partition is accepting.*

**Translation of LTL$_f$ formulas to NFA through Alternating Automata**

In [26], an efficient algorithm is presented for the conversion of LTL over finite traces (LTL$_f$) into alternating finite automata (AFA). First the LTL$_f$ formula is translated into a more expressive logic, Linear Dynamic Logic over finite traces. (LDL$_f$). This process can be done in linear time.

Then, the LDL$_f$ formula is converted into negation normal form (NNF). The resulting formula $\varphi$ is used to construct an AFA that accepts the language of $\varphi$. The number of states in $A_\varphi$ is linear in the size of $\varphi$.

**Theorem 1** (Complexity of LTL$_f$ to AFA Translation, [26])**.** *The translation of LTL$_f$ to AFA is linear in time and space with respect to the size of the input formula.*

Then in [27], the existence of an alternating finite automaton for an LTL$_f$ formula is used to present an algorithm for the translation of LTL$_f$ formulas into NFA.

**Theorem 2** (Complexity of LTL$_f$ to NFA Translation, [27])**.** *The overall translation from an LTL$_f$ formula $\varphi$ to an NFA $A_\varphi$ formula such that $\pi \models \varphi$ iff $\pi \in \mathcal{L}(A_\varphi)$ is **exponential** in time and the size of the states in $A_\varphi$ is at most exponential in the size of the original formula.*

### 2.2.4 Automata on Infinite Trees

Automata on trees generalise word automata to tree-shaped inputs and are central to automata-theoretic model checking of branching-time and strategy logics, such as ATL$^*$. In particular, Buchi Tree Automata are used to characterise the set of strategy trees for a coalition of agents.

We focus here on nondeterministic automata over infinite, full $d$-ary trees labelled over an input alphabet $\Sigma$, following the classical framework in [72, 1].

**Definition 9.** *A $d$-**ary tree** over an alphabet $\Sigma$ is a function $t : D^* \to \Sigma$, where $D = \{0, 1, \ldots, d-1\}$ and $D^*$ is the set of all finite sequences over $D$. The domain of the tree corresponds to the nodes, with the root at the empty sequence $\varepsilon$, and each node $v \in D^*$ has $d$ children at positions $v \cdot i$ for $i \in D$.*

**Definition 10.** *A **nondeterministic tree automaton** over $d$-ary trees is a tuple $A = \langle Q, \Sigma, \delta, q_0, \mathcal{F} \rangle$, where:*

- *$Q$ is a finite set of states,*
- *$\Sigma$ is the input alphabet,*
- *$q_0 \in Q$ is the initial state,*
- *$\delta : Q \times \Sigma \to Q^d$ is the transition function, mapping a state and input symbol to a set of $d$-tuples of successor states,*
- *$\mathcal{F}$ is an acceptance condition over infinite branches (e.g., Büchi, Rabin, parity).*

A **run** of $A$ on a tree $t : D^* \to \Sigma$ is a tree $r : D^* \to Q$ such that:

- $r(\varepsilon) = q_0$,
- For all $v \in D^*$, if $r(v) = q$ and $t(v) = a$, then $(r(v \cdot 0), \ldots, r(v \cdot (d-1))) \in \delta(q, a)$.

Each infinite branch of the run tree yields a sequence of states. The tree is accepted if at least one run exists such that all infinite branches of the run satisfy the acceptance condition $\mathcal{F}$, as defined previously.

## 2.3 Model Checking: Concepts and Challenges

### 2.3.1 The Model Checking Problem

Formal verification is a critical approach for ensuring the correctness, safety, and reliability of systems by rigorously proving that a system satisfies its specifications. Unlike traditional testing, which examines a limited set of scenarios, formal verification explores all possible system behaviours, providing exhaustive guarantees ([4], [47]). Within formal verification, model checking, introduced by Clarke and Emerson [22], and independently by Queille and Sifakis [62], is an automated technique that systematically analyses whether a system model satisfies certain desired properties.

Model checking relies on three core components: a formal model of the system (e.g., a Kripke structure or a Concurrent Game Structure for multi-agent systems), a specification language (usually temporal logics like LTL, CTL, or ATL), and an algorithmic procedure to verify whether the properties hold [56]. Models of the system describe its behaviour, typically through finite state automata, where a set of states and transitions between them capture how the system evolves over time [4]. To use these model checkers, the formal model is often translated into a model description language such as Promela (used by SPIN), ISPL (used by MCMAS), or SMV (used by NuSMV).

Properties about the system are described using temporal logic, which allows the expression of constraints and requirements over time. The choice of temporal logic depends on the nature of the system and the properties to be verified. The model checking algorithm systematically explores the state space of the system, often using optimisations to manage complexity, and provides rigorous guarantees about the properties of the system.

Thus, we can define the model checking problem formally, following the standard definition as in [47].

**Definition 11.** *Given a model of the system $M$ and a property $\varphi$ expressed in temporal logic, the model checking problem involves determining whether $M, s \models \varphi$, i.e. whether the property $\varphi$ holds at a given state $s$ of the model.*

### 2.3.2 State Explosion Problem

One of the primary challenges in model checking is its reliance on exhaustive exploration of the system's state space. The size of this state space grows exponentially with the complexity of the system being modelled; a phenomenon commonly referred to as the state explosion problem [56]. Traditional model checkers that use explicit state exploration can typically handle models with up to $10^8$ to $10^9$ states [4]. However, for larger and more complex systems, such as those encountered in industrial applications, this approach becomes computationally infeasible. To address this limitation, several techniques have been developed, including symbolic model checking, partial order reduction, and abstraction. In this work, we focus on symbolic model checking, which is well-suited to the ATL$^*$ model checking algorithms developed here. For completeness, we briefly summarise the other two techniques below.

**Partial Order Reduction.** Partial order reduction (POR) aims to reduce redundant interleavings of independent concurrent actions during state exploration [60]. While highly effective for

interleaving-based temporal logics like LTL, POR is less applicable to ATL*, where the semantics are based on strategic reasoning. In ATL, agent strategies are generally interdependent, making it difficult to identify independent transitions and limiting the effectiveness of POR techniques in this setting.

**Abstraction.** Abstraction reduces the state space by creating a simplified model that over-approximates the behaviour of the original system [23]. If the abstract model satisfies a property, then the concrete model does as well. However, the converse does not hold, and false negatives may arise. Techniques such as Counterexample-Guided Abstraction Refinement (CEGAR) address this by iteratively refining the abstraction when spurious counterexamples are detected [21].

### Symbolic Model Checking

Symbolic model checking was a solution presented by Mcmillan in [56] to address the state explosion problem. It avoids the explicit enumeration of all states and transitions by representing the state space and transition relations of a system compactly using Boolean functions and efficiently manipulating them with Ordered Binary Decision Diagrams (OBDDs). We define OBDDs as presented by Bryant in [13].

**Definition 12.** *An OBDD is a reduced directed acyclic graph (DAG) representing a Boolean function with $f(x_1, x_2, \ldots, x_n)$, where:*

- *Each internal node corresponds to a variable*
- *Each edge represents a true or false assignment to the variable*
- *Leaf nodes represent the function's output (true or false)*
- *The variables appear in a fixed order along all paths in the graphs.*

*To construct an OBDD, a boolean function is recursively decomposed into:*

$$f(x_1, x_2, \ldots, x_n) = x_1 \wedge f_1 + \neg x_1 \wedge f_0$$

*Where $f_1$ and $f_0$ are the subfunctions for $x_1 = 1$ and $x_1 = 0$ respectively.*

OBDDs eliminate redundancies by merging identical subgraphs and removing unnecessary nodes, significantly reducing memory usage for large functions. Boolean operations (e.g. conjunction, disjunction and negation) as well as quantifiers (e.g. existential quantification) can be performed directly on OBDDs, and efficient algorithms for composing and applying functions to OBDDs are given in [13]. A fixed variable ordering ensures that OBDDs provide a canonical representation for each boolean function, which means that equivalence checking can be reduced to constant time [4]. However, the disadvantage of OBDDs is that the size of the OBDD relies heavily on the variable ordering. For some functions, an optimal ordering can result in a linear-sized OBDD, while a poor ordering can lead to exponential growth [13]. Determining the best ordering is often a manual process that requires intuition about the system, although some heuristics for its implementation in symbolic model checkers have been proposed in [14]. Additionally, BDD libraries such as CUDD [25], which we use in this work, employ such efficient variable ordering heuritiscs to avoid the worst-case blow-up.

Symbolic model checking using OBDDs has been widely adopted due to its scalability for verifying systems with very large state spaces, and it plays a central role in the symbolic algorithms presented in this work.

## 2.4 Literature Review

### 2.4.1 Overview of Model Checking Tools

Model checking tools have been developed to address a wide range of verification challenges, from hardware design to distributed systems and multi-agent interactions. Prominent model checkers like SPIN and MCMAS cater to different domains and methodologies, leveraging some of the techniques previously discussed such as explicit-state exploration, symbolic representations and the use of various temporal logics for the specification of properties. We provide an overview of these tools, highlighting their core features, applications and contributions to formal verification.

The SPIN model checker is a widely used tool for verifying distributed systems and communication protocols, developed by Gerard Holzmann at Bell Labs. It employs explicit-state exploration to detect errors such as deadlocks and race conditions in concurrent systems. SPIN uses Promela (Process Meta Language) to model system behaviour, and properties to be expressed are in Linear Temporal Logic (LTL), which SPIN translates into Büchi automata for verification. The verification process involves computing the synchronous product of the Büchi automaton representing the converse property and the automaton representing the system. Model checking then reduces to checking whether the product automaton is empty - i.e. whether no behaviours in the system satisfy the converse property. SPIN's efficiency is enhanced by techniques like partial order reduction and state compression to manage memory usage [43]. Over the years, SPIN has been used to verify a wide range of systems, and more recently, it was applied to analyse and identify flaws in the ADAPRO C++ software framework developed at CERN [52].

While SPIN demonstrates the strengths of explicit-state model checking in analysing concurrent systems and protocols, the MCMAS model checker is a specialised tool that focuses on symbolic techniques for the verification of multi-agent systems (MAS), supporting the analysis of temporal, epistemic, and strategic properties. MCMAS models MAS using the Interpreted Systems Programming Language (ISPL), which allows the specification of agents, their local states, protocols and their interactions. It uses symbolic model checking techniques with OBDDs to efficiently manage large state spaces, and fix point computations to iteratively compute the set of states where the required property holds. MCMAS supports properties expressed in CTL, ATL and ATLK (ATL with epistemic operators). Over the years, MCMAS has been applied to diverse scenarios, such as verifying authentication protocols to detect vulnerabilities like man-in-the-middle attacks [53].

Due to its complexity there are currently **no model checkers specifically for ATL***. However, model checking algorithms for more expressive logics like Strategy Logic (SL), which strictly subsumes ATL*, have had practical implementations. Strategy logic (SL) allows explicit quantification and binding of strategies to agents, enabling the specification of complex game-theoretic properties, such as Nash equilibria, which are not expressible in ATL* [19]. The model checker MCMAS-SLK supports SL and its extension SLK (Strategy Logic with Knowledge), which adds epistemic reasoning capabilities. This tool allows verification of SLK properties by combining symbolic model checking techniques with epistemic modalities to synthesise strategies for agents under imperfect information [16]. By supporting SL, MCMAS-SLK can also verify ATL* formulas as a subset of SL specifications. However, the enhanced expressiveness of SL comes at the cost of higher computational complexity, making specialised algorithms to verify only ATL* potentially more efficient in practice.

Furthermore, approximation methods have also been proposed to address the lack of ATL* model

checkers. These methods translate ATL* formulas into ATL and use ATL model checkers such as MOCHA [65] but they provide only approximate results, yielding strong and weak bounds rather than precise verification. Additionally, certain subsets of ATL* cannot be translated into ATL due to ATL*'s strictly greater expressiveness, limiting the effectiveness of such approximations [40].

### 2.4.2 Model Checking Complexity

The computational complexity of model checking varies significantly depending on the logic used and its expressive power. Table 2.2 summarises the complexities of model checking for some key logics under the assumption of perfect recall strategies.

**Table 2.2:** Model Checking Complexity of Various Logics

| **Logic** | **Complexity** | **Reference** |
|---|---|---|
| ATL | PTIME | [1] |
| ATL* | 2EXP-TIME-complete | [1] |
| $ATL^*_f$ | 2EXP-TIME-complete | [7] |
| Strategy Logic (SL) | Non-elementary | [57] |
| One-alternation fragment of SL | 2EXP-TIME-complete | [19] |
| One-goal fragment of SL | 2EXP-TIME-complete | [57] |

The results highlight the trade-off between expressiveness and computational complexity. For example, pure Strategy Logic (SL) offers unmatched expressiveness but suffers from non-elementary complexity, limiting its applicability for model checking ATL* to small systems. However, the one-alternation and one-goal fragments of SL, which are double exponential, match the complexity of model checking ATL* over finite traces.

# Chapter 3

# Finite Trace Model Checker Development

This chapter details the development of a model checker of ATL$^*_f$, including both an explicit-state implementation and a novel symbolic algorithm and implementation. Building on the theoretical framework proposed in [7], this work presents the first complete model checking tool for ATL$^*_f$, capable of verifying whether an input ATL$^*$ formula holds over a finite-trace interpretation of a multi-agent system specified in ISPL.

The development began with an explicit-state version, using concrete data structures to represent both the system and the automata derived from temporal subformulas. To improve scalability and reduce memory overhead, a symbolic model checker was designed. The symbolic version reshapes the algorithm around Binary Decision Diagrams (BDDs), requiring new design decisions for state encoding and formulas for fixpoint computation, while preserving the recursive evaluation strategy.

Both implementations share a common modular architecture, including an ISPL parser, integration of external tools for translating LTL$_f$ subformulas into DFAs, construction of the synchronous product between the system and the automaton, and a fixpoint-based procedure to solve the strategic safety game defined by the formula. These components are fitted within a recursive model checking procedure that evaluates ATL$^*_f$ by structural decomposition. Figure 3.1 illustrates this process across four stages: input, formula decomposition, the solution to the strategic formula through DFA and product contruction, and game solving, and the final evaluation to determine satisfiability.

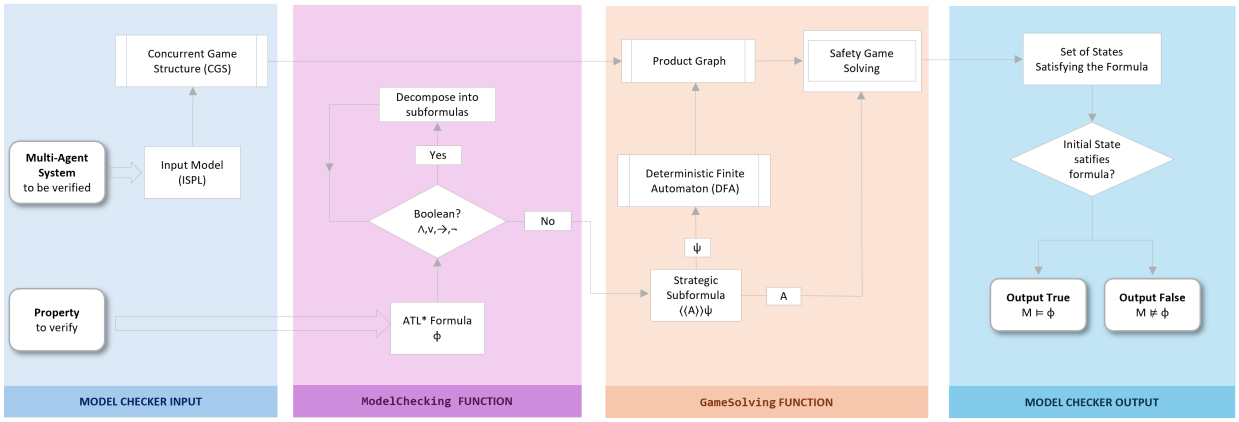


**Figure 3.1:** Overview of the modular architecture and main components of the ATL$^*_f$ model checking process

This chapter begins with a description of the system input, followed by the implementation details of the explicit and symbolic versions respectively. The performance of both implementations is evaluated comparatively in Chapter 5.

## 3.1 Multi-Agent System Input

The model checker accepts system specifications written in ISPL (Interpreted Systems Programming Language), a high-level language for synchronous multi-agent systems originally developed for the MCMAS model checker [53]. ISPL was selected to enable potential future integration with the MCMAS ecosystem and to leverage existing modelling conventions for agent-based systems. It offers a flexible syntax for describing agents, protocols, local state variables, and an environment process to capture system dynamics.

For this project, a subset of ISPL has been implemented, tailored specifically for ATL$^*_f$ model checking over Concurrent Game Structures (CGSs). This design aligns naturally with the semantics of CGSs, which describe global-state transitions based on joint actions rather than individual local states or agent-specific transitions. As such, certain ISPL features are restricted or interpreted differently:

- **Boolean variables:** The current implementation supports only Boolean variables. While standard ISPL allows integers and enumerated types along with arithmetic expressions, these can be incorporated in future extensions without altering the underlying CGS semantics.
- **No explicit local agent states:** Local state declarations are omitted, as CGSs do not model agent-local states explicitly. Instead, agent protocols are interpreted over global conditions, in line with CGS-based reasoning.
- **Global evolution:** Transitions are defined through the environment's global `Evolution` section. Agent-specific evolution blocks, which are meaningful in interpreted systems semantics, are excluded here to maintain consistency with the CGS model.

A central extension introduced in this project is the ability to specify ATL* formulas within the ISPL input. While standard ISPL supports only ATL and epistemic logic (ATLK), it lacks support for the full ATL* syntax. To address this, the `Formulae` section is extended with a custom recursive descent parser that has been implemented to translate ATL* formulas into an internal representation used by the model checker. Additionally, a `FinalStates` section has been added to explicitly declare the final states of the system, which are required to support the finite trace semantics of ATL$^*_f$.

Currently, syntactic validation is performed for ATL* formulas, while validation of other ISPL sections remains limited.

## 3.2 Explicit-State Model Checker

The explicit-state model checker provides a concrete implementation of the ATL$^*_f$ algorithm using data structures that enumerate all reachable system states and transitions. The system model is parsed into a Concurrent Game Structure (CGS), internally represented using the Boost Graph Library (BGL). Global states form the vertices of the graph, and edges are labelled with joint actions over all agents. The use of BGL's `adjacency_list` with `vecS` for both vertices and edges enables

compact storage and efficient traversal during product construction and fixpoint evaluation [10]. Proposition valuations and final state sets are encoded as `boost::dynamic_bitsets` to support compact and efficient set operations. The remainder of this section describes the architecture and implementation of each core component.

### 3.2.1 `GameSolving` Algorithm

The `GameSolving` Algorithm is the core component of the model checker responsible for evaluating strategic subformulas of the form $\langle\langle A \rangle\rangle \psi$, where $\psi$ is an LTL$_f$ formula. Given a CGS and such a formula, the algorithm determines the set of states from which the coalition $A$ has a strategy to ensure that $\psi$ is satisfied on all resulting execution paths.

The algorithm proceeds in several stages. First, the linear temporal goal $\psi$ is converted into a Deterministic Finite Automaton (DFA), which accepts all finite traces that satisfy the formula. Then, a synchronous product is constructed between the CGS and this DFA, capturing all valid system executions and the corresponding progress of the formula. From this product, a safety game is derived and solved via a greatest fixpoint computation to determine the coalition's winning region.

#### DFA Construction

The first stage of the algorithm involves translating the LTL$_f$ formula $\psi$ into a DFA. To ensure robust and efficient performance across a range of input formulas, the implementation adopts a parallelised translation strategy, which integrates three external tools—Lydia, Lisa2 and LTLf2DFA—all of which generate DFAs for LTL$_f$ formulas.

To leverage the complementary performance characteristics of these tools, they are executed asynchronously and in parallel. The output of whichever tool completes first is used, while the other processes are terminated. This design choice is based on empirical evaluation results (discussed in Appendix A), which show that each tool performs better on different classes of formulas. By exploiting this parallel execution model, the overall latency of DFA construction is reduced in many cases.

#### Tool Outputs and Parsing Challenges

Lydia and LTLf2DFA output the resulting automaton in Graphviz `.dot` format. A custom parser was implemented to convert this representation into the internal graph structure used by the model checker. Lisa2, in contrast, generates automata in the Hanoi Omega Automata (HOA) format, typically used to describe Büchi automata. Although the automaton produced is deterministic, its structure is defined using Büchi acceptance conditions. In particular, Lisa2 introduces a special atomic proposition `alive`, which appears in the transition labels of all non-terminal transitions. A designated sink state, reachable via transitions labeled with $\neg$`alive`, encodes the termination of the trace. The custom HOA parser identifies all states that have an outgoing transition to this sink state as the accepting (final) states of the DFA, aligning with the intended semantics of finite trace acceptance.

**Transition Label Representation**

Once the DFA corresponding to the $\text{LTL}_f$ formula is parsed into an internal BGL graph structure, a key challenge arises in interpreting its transitions. **Figure 3.2** illustrates an example DFA output by the external tools. Each transition is guarded by a vector over atomic propositions, where each position may specify a required value (0 or 1) or a don't-care symbol (X). These labels are interpreted as Disjunctive Normal Form (DNF) formulas and must be checked against the proposition assignment of the CGS state during product construction.

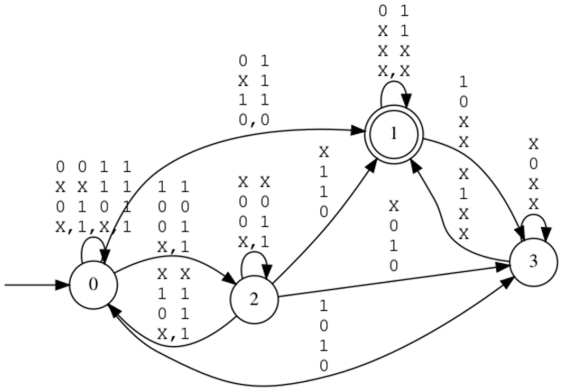


**Figure 3.2:** Example DFA for the formula $G(a \rightarrow Fb) \wedge F(c \wedge \neg b)$

This check is performed repeatedly and at scale, both in the breadth-first construction of the product graph and later in the fixpoint computations. Therefore, the structure used to represent and evaluate these transition labels must be efficient in both space and time, to avoid performance bottlenecks.

Several approaches were considered. Direct DNF representations are intuitive and compact but rely on set comparisons that are difficult to optimise. Binary Decision Diagrams (BDDs) offer compactness and reuse but introduce overhead and complexity for the small formulas typically encountered. SAT-based approaches, involving CNF encoding and solver invocations, were also explored but rejected due to their high runtime overhead and integration cost.

The adopted solution encodes each transition label as a binary vector paired with a mask, where each bit corresponds to an atomic proposition. The label vector specifies required values (with `X` values in don't-care positions), while the mask indicates which positions are relevant. The set of propositions true at a state is likewise represented as a bit vector. The satisfiability check then reduces to a bitwise masking and comparison: if the masked assignment matches the masked label, the transition is satisfied.

This representation is compact, efficient, and integrates naturally with the existing `boost::dynamic_bitset`-based infrastructure. In addition to constant-time evaluation, it supports straightforward parallelisation via SIMD techniques (e.g., AVX2), making it well suited to large-scale model checking tasks. While storing both a label and a mask incurs some overhead, the trade-off yields strong practical performance across a broad range of DFA encodings.

### Product Graph Construction

Once the CGS and the DFA corresponding to the LTL$_f$ formula are available, the next step is to construct their synchronous product. The product graph encodes all valid system executions along with the corresponding progress through the temporal formula, and forms the basis for solving the strategic game.

Each node in the product graph is a pair $(s, q)$, where $s$ is a state of the CGS and $q$ is a state of the DFA. The initial node is given by $(s_0, \delta(q_0, \lambda(s_0)))$, where $\lambda(s_0)$ denotes the set of propositions true in the initial CGS state and $\delta$ is the DFA transition function. The product graph is constructed using a breadth-first search (BFS), restricted to reachable states only, in order to avoid unnecessary exploration of unreachable regions of the state space.

A practical challenge encountered during this stage arises from the structure of the DFAs generated by Lisa2, which do not explicitly include transitions for every possible valuation of propositions. This creates a problem during product graph construction: when moving from a CGS state $s$ to a successor $s'$, the product construction requires a corresponding DFA transition from the current automaton state $q$ on the valuation $\lambda(s')$. If no such transition exists, the construction would become incomplete, and would affect the correctness of the fixpoint solution. To handle this, the implementation does not modify the DFA itself by adding missing transitions since checking whether every single evaluation is mapped would incur a high overhead. Instead, it introduces a single global sink node into the product graph, which is only used if a DFA transition is missing, in which case the product graph includes a transition to the sink node $(s', \text{sink})$. The sink node represents failure to satisfy the temporal formula and is treated as a non-accepting, absorbing state. This approach ensures that all product transitions are well-defined, while avoiding unnecessary overhead and ensuring that processing latencies are similar between outputs from all three tools.

### Safety Set and Fixpoint Computation

Once the product graph is constructed, the final step of the `GameSolving` algorithm is to identify the set of states from which the coalition can ensure satisfaction of the temporal goal $\psi$. This is formulated as a safety game over the product graph and solved using a greatest fixpoint computation.

The first step is to define the safety set. A product state $(s, q)$ is safe if it satisfies one of two conditions: the CGS state $s$ is not marked as final (i.e., the execution has not yet terminated), or the CGS state $s$ is final and the DFA state $q$ is also accepting.

To compute the set of winning states for the coalition, a greatest fixpoint is computed over the product graph. At each iteration, the current set $Y$ is intersected with its predecessor set:

$$Y \leftarrow Y \cap \text{Pre}(Y),$$

where $\text{Pre}(Y)$ contains all states from which the coalition has a joint action such that every possible successor (under all completions by non-coalition agents) remains in $Y$.

A key challenge in this step is the efficient evaluation of the predecessor operator. While the product graph is represented using an adjacency list of nodes and outgoing edges, computing $\text{Pre}(Y)$ requires identifying, for each coalition action at a node, the full set of resulting successors. A naive approach would involve re-computing this set repeatedly for every node in every iteration, which

would be prohibitively expensive.

To address this, the implementation builds an auxiliary index during product graph construction. For each product node, it records, for each coalition action, the set of successors reachable under all completions of that action. This preprocessing step takes $O(|E|)$ time, but allows constant-time access during the fix-point computation, significantly reducing overall runtime.

The fixpoint iteration continues until convergence. The resulting set of winning states is projected onto the CGS component to determine the states where the coalition can enforce the formula, i.e. the states $s$ for which $M, s \models \langle\langle A \rangle\rangle \psi$ holds.

### 3.2.2 `ModelChecking` Algorithm

The `ModelChecking` function performs recursive labelling of the input ATL$^*_f$ formula based on its syntactic structure. Each formula node is handled according to its type, which may be either a propositional operator (e.g., $\neg$, $\wedge$, $\vee$) or a coalition operator of the form $\langle\langle A \rangle\rangle \psi$.

Propositional operators are evaluated recursively using their standard semantics. For example, a conjunction is satisfied in the intersection of the states satisfying its operands, and negation is interpreted as set complement. These cases are implemented directly as recursive calls over the subformulas.

When the formula is of the form $\langle\langle A \rangle\rangle \psi$, the algorithm first extracts all maximal state subformulas from $\psi$. These are evaluated and replaced by fresh atomic propositions, which are then marked as true in precisely the states where the corresponding subformulas hold. This ensures that $\psi$ becomes an LTL$_f$ formula, suitable for DFA construction and evaluation using the `GameSolving` algorithm described in Section 3.2.1.

Formulas are internally represented using a recursive data structure that mirrors the design of formulas in the MCMAS tool [53]. Each formula is stored as a node containing an operator tag and up to two operands, represented as shared pointers to subformulas. This representation is concise and flexible, allowing for efficient traversal and transformation during evaluation.

## 3.3 Symbolic Model Checker

While the explicit-state implementation of the ATL$^*_f$ model checker provides a faithful realisation of the algorithm, explicit techniques are inherently limited in their ability to scale to large or complex systems. As the number of states grows, so does the cost of storing and traversing them, quickly leading to performance bottlenecks. To address these challenges, this chapter introduces a symbolic variant of the model checker, designed to mitigate state explosion through the use of Binary Decision Diagrams (BDDs). Although symbolic model checking is a well-established technique, this work presents the first symbolic formulation and implementation of the automata-theoretic algorithm for verifying ATL$^*_f$ formulas.

The symbolic model checker adopts a semi-symbolic approach: the Concurrent Game Structure (CGS) is encoded symbolically, while the Deterministic Finite Automaton (DFA) representing the LTL path formula $\psi$ remains explicit. This design reflects the fact that CGSs typically have much

larger state spaces than the formulas being verified, making symbolic encoding more beneficial on the model side. Moreover, as DFA construction is handled by external tools and often dominates the formula-related cost, a symbolic encoding of the DFA would offer limited advantage.

The overall structure of the model checker is preserved. The recursive labelling algorithm used to evaluate the ATL$^*_f$ formulas remains conceptually identical to the explicit implementation, including the decomposition of temporal and Boolean operators. The key modification lies in the symbolic implementation of the `GameSolving` procedure, which replaces its explicit counterpart and is detailed in the remainder of this chapter.

### 3.3.1 Symbolic Encoding of the CGS

The symbolic encoding of the CGS is based on BDDs. Each structural component of the CGS—states, actions, transition relation, and labelling function—is encoded as a BDD to enable scalable manipulation and fixpoint computation. This section describes the encoding of each component.

The implementation uses the CUDD library [25], which provides efficient operations for BDD construction and manipulation, including conjunction, disjunction, existential quantification, and dynamic variable reordering. Its performance and widespread use in symbolic verification make it well-suited for this use case.

#### State Encoding

Let $S$ denote the set of states in the CGS, with $|S| = N$. Each state $s \in S$ is represented as a binary vector of length $n = \lceil \log_2 N \rceil$, using Boolean variables $\vec{q} = (q_0, q_1, \ldots, q_{n-1})$. Thus a single state $s_i$ corresponds to a unique conjunction over these variables. For example, the state $s_3$ in a 3-bit encoding corresponds to the assignment $q_0 \wedge q_1 \wedge \neg q_2$. Sets of states are then naturally represented as disjunctions of such conjunctions and encoded compactly as BDDs. Bit-level encoding is used instead of one-hot encoding to minimise the number of Boolean variables, thereby improving BDD efficiency and scalability.

To describe transitions, a second copy of the state variable vector $\vec{q'} = (q'_0, q'_1, \ldots, q'_{n-1})$ is introduced to represent the destination state in a transition.

#### Action Encoding

Let $A_1, A_2, \ldots, A_m$ be the agents in the system, and let agent $A_i$ have a set of available actions $\text{Act}_i$ with $|\text{Act}_i| = N_i$. The action of each agent is encoded using $n_i = \lceil \log_2 N_i \rceil$ Boolean variables, denoted $\vec{a_i} = (a_{i,0}, a_{i,1}, \ldots, a_{i,n_i-1})$. A global joint action is therefore represented using the union of all agent's action variables:

$$\vec{a} = \bigcup_{i=1}^{m} \vec{a_i}$$

#### Transition Function Encoding

A transition $\delta(s, \alpha, s')$, where $\alpha \in \text{Act}_1 \times \cdots \times \text{Act}_m$, is represented by a conjunction over:

- current state variables $\vec{q}$

- joint action variables $\vec{a}$
- next state variables $\vec{q'}$

The entire transition relation $\delta \subseteq S \times \text{Act}_1 \times \cdots \times \text{Act}_m \times S$ is then encoded as a BDD defined as the disjunction of BDDs for individual transitions. Symbolically:

$$\delta(\vec{q}, \vec{a}, \vec{q'}) = \bigvee_{(s,\alpha,s') \in \delta} [\text{enc}_\text{c}(s) \wedge \text{enc}_\text{a}(\alpha) \wedge \text{enc}_\text{n}(s')]$$

where $\text{enc}_\text{c}$, $\text{enc}_\text{n}$ denote the Boolean encoding of a state over the current or next states respectively, and $\text{enc}_\text{a}$ is the encoding of a joint action.

#### Labelling Function and Propositions

The atomic proposition labelling function $\lambda : S \to 2^{AP}$ is inverted to suit symbolic processing. Specifically, we define $\lambda' : AP \to \mathcal{B}(\vec{q})$, where $\mathcal{B}(\vec{q})$ denotes BDDs over state variables, and $\lambda'(p)$ gives the set of states in which proposition $p$ holds. This reformulation allows us to substitute propositions in LTL or DFA transition labels with the BDDs of the corresponding states, facilitating the symbolic product computation described in the next section.

The labelling function is implemented as a mapping from each atomic proposition $p \in AP$ to a BDD representing the set of states in which $p$ is true.

#### Initial and Final States

The initial state $s_0 \in S$ is encoded as a BDD over $\vec{q}$ corresponding to a single valuation. The set of final states $F \subseteq S$ is similarly represented as a BDD over $\vec{q}$, supporting efficient set operations during fixpoint computations and safety checks.

### 3.3.2 Product Graph Computation

To verify a formula of the form $\langle\langle A \rangle\rangle \psi$, where $\psi$ is an LTL formula, we construct a symbolic representation of the product between the CGS and the corresponding DFA for $\psi$. The CGS is encoded symbolically using BDDs, as described in Section 3.3.1, while the DFA remains explicit but is translated into symbolic form during product computation.

This section outlines the symbolic construction of the product transition relation and incorporates essential components such as sink state handling (required for correctness in automata generated by tools like Lisa2) and reachability pruning to improve symbolic performance.

#### DFA Transition Representation

Each DFA transition is labelled with a propositional formula in Disjunctive Normal Form (DNF) over atomic propositions. To evaluate such transitions symbolically, we reinterpret these formulas as Boolean functions over the CGS state variables.

Given the symbolic labelling function $\lambda' : AP \to \mathcal{B}(\vec{q})$, each literal in a DFA label is replaced as follows:

- A positive literal $p$ is replaced with the BDD $\lambda'(p)$
- A negative literal $\neg p$ is replaced with $\neg\lambda'(p)$

These replacements yield a BDD that captures the set of CGS states satisfying the DFA transition label.

However, due to the semantics of the product construction, the truth valuation of atomic propositions must be determined in the destination CGS state, not the source. Therefore, the resulting BDD is composed over the next-state variables $\vec{q'}$, rather than the current-state variables $\vec{q}$. This is achieved through a variable substitution using a `VectorCompose` operation that maps each $q_i$ to $q'_i$.

**Variable Setup for DFA encoding**

Let $D$ be the set of DFA states with $|D| = M$. We introduce:

- A set of Boolean variables $\vec{d} = (d_0, \ldots, d_{m-1})$ with $m = \lceil \log_2 M \rceil$ to represent the current DFA state, and
- A corresponding set $\vec{d'}$ to represent the next DFA state.

Each DFA transition $d \xrightarrow{\phi} d'$ is thus represented as a conjunction over:

- $\text{enc}_\text{c}(d)$ over $\vec{d}$,
- the BDD for the translated formula $\phi$ over $\vec{q'}$, and
- $\text{enc}_\text{n}(d')$ over $\vec{d'}$.

The full DFA transition relation $\Delta$ becomes a BDD over $(\vec{d}, \vec{q'}, \vec{d'})$.

**Coalition Abstraction**

The symbolic CGS transition relation $\delta(\vec{q}, \vec{a}, \vec{q'})$ encodes all possible transitions for all joint actions. However, when verifying a coalition $A \subseteq$ Agents, we are interested only in those transitions that can be enforced by some joint action of the coalition.

To account for this, we existentially quantify over the action variables of agents not in $A$. Let $\vec{a}_{-A}$ denote these non-coalition action variables. Then:

$$\delta_A(\vec{q}, \vec{a}_A, \vec{q'}) = \exists \vec{a}_{-A}.\delta(\vec{q}, \vec{a} \wedge \vec{a}_{-A}, \vec{q'})$$

which yields a BDD that captures all transitions that the coalition $A$ can enforce.

**Product Transition Relation**

The final symbolic transition relation of the product is defined over the combined variable set $(\vec{q}, \vec{a}_A, \vec{q'}, \vec{d}, \vec{d'})$ as:

$$\delta'(\vec{q}, \vec{a}_A, \vec{q'}, \vec{d}, \vec{d'}) = \delta_A(\vec{q}, \vec{a}_A, \vec{q'}) \wedge \Delta(\vec{d}, \vec{q'}, \vec{d'})$$

This BDD encodes all valid transitions in the product system from a pair of CGS and DFA states under coalition actions.

#### Sink Handling for Missing Transitions

As described in Section 3.2.1, product construction requires a special handling of DFA states generated by Lisa2 with incomplete transition coverage. In contrast to the explicit setting where this is handled directly in the product, in the symbolic setting this must be addressed during the translation of the DFA to BDDs.

For each DFA state $d$, we compute the disjunction of the BDDs of all its outgoing transition labels. If this disjunction does not cover the full Boolean space (i.e. its negation is not the zero function), we add a synthetic transition from $d$ to a special sink state $d_{\text{sink}}$ with the negated BDD as guard. This ensures totality of the DFA transition function in the symbolic product while avoiding the need to inspect individual CGS states explicitly.

#### Reachability Pruning

To avoid unnecessary operations over unreachable product states, we apply a symbolic reachability analysis after constructing the product transition relation. The reachability set is computed using a standard fixpoint algorithm.

1. Let $R_0$ be the BDD for the initial product state $(s_0, \Delta(d_0, \lambda(s_0)))$
2. At each iteration $i$, compute the set of successors:

$$R_{i+1} = R_i \vee \text{Post}(R_i),$$

   where $\text{Post}(R)$ is computed by conjoining $R$ with the transition relation $\delta'$, then projecting the next-state variables $(\vec{q'}, \vec{d'})$ back to current-state variables $(\vec{q}, \vec{d})$ via `VectorCompose`.
3. Repeat until convergence : $R_{i+1} = R_i$

The final reachable set $R$ is then used to trim the transition relation $\delta'$ to only include transitions between reachable states.

#### Safety Set for Fixpoint Solving

The symbolic safety objective is encoded as

$$Y_0 = R \wedge (\neg F \vee F'),$$

where $R$ is the set of reachable product states, $F$ is the BDD for the CGS final states, and $F'$ is the BDD for the final (accepting) states in the DFA. This set defines the initial domain of states from which the safety fixpoint algorithm (described in Section 3.3.3) will operate.

### 3.3.3 Symbolic Fixpoint Computation

Following the construction of the symbolic product between the CGS and the DFA, the verification of a formula of the form $\langle\langle A \rangle\rangle \psi$ reduces to solving a safety game over the product transition relation. The objective is to compute the set of product states from which coalition $A$ can ensure staying within a designated safe set. This is done via a fixpoint computation using a symbolic pre-image operator. Let $Y \subseteq S \times D$ denote the current set of safe product states, encoded as a BDD over the current-state variables $(\vec{q}, \vec{d})$. The symbolic pre-image of $Y$ under the product transition relation $\delta'$ is computed as follows:

1. Encode unsafe successors by negating $Y$ over next-state variables to obtain $\neg Y'$.
2. Identify transitions to unsafe states:
$$T_{bad}(\vec{q}, \vec{a}_A, \vec{q'}, \vec{d'}) = \delta' \wedge \neg Y'.$$
3. Abstract away next-state variables to obtain the states from which there is an action that leads to an unsafe state:
$$B(\vec{q}, \vec{a}_A) = \exists \vec{q'}, \vec{d'}.T_{bad}$$
4. Negate to identify safe transitions, states in which there is an action that leads only to safe states:
$$G(\vec{q}, \vec{a}_A) = \neg B$$
5. Quantify over coalition actions to obtain the pre-image:
$$\text{Pre}(Y)(\vec{q}, \vec{d}) = \exists \vec{a}_A.W$$

The fixpoint is computed by repeatedly applying the pre-image operatior start from the initial safety set $Y_0$ and iterating:
$$Y_{i+1} = Y_i \wedge \text{Pre}(Y_i)$$
until convergence: $Y_{i+1} = Y_i$. The resulting BDD $Y$ represents the set of product states from which coalition $A$ can enforce satisfaction of $psi$. To extract the CGS states satisfying $\langle\langle A \rangle\rangle \psi$, we perform the following:

1. Conjoin transitions from $d_0$ with the fixpoint set $Y$, encoded over next-state variables
2. Abstract away both current and next DFA variables
3. Use `VectorCompose` to map the next-state CGS variables to current-state variables.

This resulting BDD, encoding the states satisfying $\langle\langle A \rangle\rangle \psi$, is then used in the recursive model checking procedure exactly as in the explicit implementation, detailed in Section 3.2.2.

# Chapter 4

# Infinite Trace Model Checker Development

Although the model-checking problem for ATL* is 2EXPTIME-complete over both finite and infinite traces, practical performance often diverges: finite-trace procedures manipulate word automata, whereas infinite-trace procedures must handle $\omega$-automata, with inherently more complex acceptance conditions [7]. We therefore first implemented a finite-trace ATL* model checker, anticipating that its simpler intermediate automata would admit a more efficient engine despite the same worst-case complexity. To determine whether this expectation holds in practice, however, it is essential to develop and evaluate a complementary infinite-trace verifier.

Moreover, certain systems such as non-terminating reactive processes or protocols with liveness guarantees cannot be soundly captured by any fixed bound on trace length. Some examples of such systems are autonomous vehicles, where self-driving cars in a city must continuously cooperate to avoid crashes, or algorithmic trading systems whose never-ending interaction with the market could be verified to ensure that undesirable states never occur, regardless of the strategies followed by other agents. For these, full infinite-trace semantics of ATL* are required, and so an ATL* model checker is not just a benchmarking tool but a necessity for broader applicability. While it is possible to reduce ATL* model checking to more expressive frameworks—e.g. Strategy Logic (SL) or SL with one-goal specifications (SL[1G]), each of which strictly subsumes ATL* [57]—the extra generality of these logics may incur significant algorithmic overhead that is avoidable when one only needs ATL*. A dedicated infinite-trace ATL* algorithm thus stands to offer both conceptual simplicity and performance benefits.

In this chapter, we present two algorithmic approaches for model checking ATL* over infinite traces. The first follows the classic construction of alternating tree automata over Rabin acceptance, as in the seminal work of Alur et al. [1]. To make this construction more tractable in practice, we introduce a hybrid symbolic-explicit implementation. The second approach reframes the verification problem as the solution of a parity game between the coalition and its complement. This algorithm provides a new formulation of ATL* model checking that avoids Rabin acceptance altogether, enabling tighter integration with symbolic methods and the use of modern parity game solvers. We detail both constructions and discuss the design decisions behind their symbolic implementations. Both implementations are evaluated in Chapter 7.

## 4.1 Rabin Tree Automata Approach

In this infinite-trace procedure, we keep the same recursive labelling scheme used in the finite-trace model checker. The difference lies when we encounter a strategic formula of the form $\langle\langle A \rangle\rangle \psi$, where we replace the finite-trace automaton for an $\omega$-automaton to represent $\psi$. In particular, to decide

$$q \models \langle\langle A \rangle\rangle \psi$$

for each state $q \in Q$, we follow the original NRTA construction and non-emptiness of Alur et al. [1] in four steps:

1. **Execution-tree automaton**. Construct a nondeterministic Büchi tree automaton $\mathcal{A}_{G,q,A}$ whose accepted trees are exactly the execution trees rooted at $q$ under all strategies of $A$.

2. **Specification automaton**. Translate the LTL formula $\psi$ into a Deterministic Rabin Tree Automaton (DRTA) $A_\psi$ that accepts precisely those infinite paths satisfying $\psi$. We use the tool Rabinizer4 [49] in this construction, since it employs a Safra-less method that generates smaller DRAs than classic approaches. We then apply a lifting construction to obtain a tree automaton from the word automaton generated by the tool.

3. **Product NRTA**. Form the synchronous product

   $$\mathcal{A}_{\text{prod}} = \mathcal{A}_{G,q,A} \otimes \mathcal{A}_\psi$$

   yielding a nondeterministic Rabin tree automaton whose language is non-empty exactly when $A$ can enforce $\psi$ from $q$.

4. **Decide emptiness of $\mathcal{A}_{\text{prod}}$** via the Kupferman-Vardi algorithm [50]. $L(\mathcal{A}_{\text{prod}}) \neq \emptyset$, then $q \models \langle\langle A \rangle\rangle \psi$. To do this we construct a Weak Rabin Alternating Automaton from the NRTA, apply the construction in [50] to generate a Weak Alternating Automaton and use the algorithms defined in [51] to solve its non-emptiness.

This algorithm is reworked into a hybrid approach. To mitigate state-space blow up, we carry out steps 1-3 symbolically, encoding both the CGS transition relation and the RTA using BDDs, in order to compute the execution-tree BTA and the product efficiently. However, since the Kupferman-Vardi emptiness test requires dualising the transition function - a transformation that resists efficient BDD encoding - we extract an explicit representation of the product automaton for step 4 and perform the non-emptiness check explicitly on this structure. In the following section we provide the symbolic encoding for the tree automata used in the first 3 steps, and the details of the symbolic re-formulations of the steps above are deferred to Appendix B.

### 4.1.1 Symbolic Encoding of Fixed-Degree Tree Automata

All three automata in our pipeline, the execution-tree Büchi automaton, the Rabin automaton for the specification, and their product NRTA, share the same fixed-degree $d$-ary structure. Concretely, each has a transition function of the form

$$\delta \;:\; Q \times \Sigma \;\longrightarrow\; 2^{Q^d},$$

which we encode symbolically using Binary Decision Diagrams (BDDs) as follows:

- **State variables.** Let $n = \log_2 |Q|$, introduce $n$ Boolean variables $\vec{q} = (q_0, \ldots, q_{n-1})$ to encode the current state and $d$ distinct copies $\vec{q^0}, \ldots, \vec{q^{d-1}}$ for the successor-state slots.
- **Proposition variables.** Since transitions are labeled by arbitrary Boolean combinations of atomic propositions, we assign one BDD variable $p_j$ per proposition $P_j$. The full proposition variable vector is $\vec{p} = (p_0, \ldots, p_{V-1})$ when there are $V$ propositions.
- **Transition relation.** We represent $\delta$ as a single BDD over

  $$\vec{q},\ \vec{p},\ \vec{q^0}, \ldots, \vec{q^{d-1}},$$

  whose satisfying assignments enumerate all valid tuples $(q, a) \to \langle q^0, \cdots, q^{d-1} \rangle$ in the automaton's transition relation.

Encoding all three automata under this uniform scheme allows us to perform boolean operations symbolically and on-the-fly during the product construction.

### 4.1.2 Theoretical Complexity

Although ATL$^*$ model checking is known to be 2-EXPTIME–complete [1], we extract concrete bounds here to enable a fine-grained comparison with our finite-trace and parity-game approaches, which share the same complexity class. Concretely, let $G$ be a CGS of size $|G|$ and let $\psi$ be an LTL formula of length $|\psi|$. Then:

1. Transforming the LTL formula $\psi$ into a DRA yields in the worst case a DRA with

   $$N \ = \ 2^{2^{|\psi|}} \text{ states}, \quad K \ = \ 2^{|\psi|} \text{ Rabin pairs.}$$

2. Embedding into a full $d$-ary tree automaton incurs only linear overhead, so the DRTA also has $N$ states and $K$ pairs.
3. The synchronous product with the execution-tree BTA (size $|G|$) yields an NRTA of

   $$n = |G| \times N \text{ states}, \quad K \text{ pairs.}$$

4. Solving the non-emptiness of an NRTA with $n$ states and $k$ pairs takes

   $$O\big(n^{2K+1} \cdot K!\big)$$

   [50].
5. Since we solve emptiness separately for each of the $|G|$ source states, the overall cost is multiplied by $|G|$.

Substituting $n = |G| \cdot 2^{2^{|\psi|}}$ and $K = 2^{|\psi|}$ gives the explicit time complexity

$$O\Big(|G| \ \cdot \ \big(|G| \cdot 2^{2^{|\psi|}}\big)^{2 \cdot 2^{|\psi|}+1} \ \cdot \ \big(2^{|\psi|}\big)!\Big) = O\Big(|G|^{2^{|\psi|+1}+2}\ 2^{\,2^{|\psi|}\,(2^{|\psi|+1}+1)}\ (2^{|\psi|})!\Big).$$

This bound is polynomial in $|G|$ with degree exponential in $|\psi|$, and doubly exponential in $|\psi|$, confirming the overall 2-EXPTIME complexity.

## 4.2 Parity Game Approach

As an alternative to the Rabin-automata pipeline, we reduce ATL* model checking to solving a parity game, which is a two-player game of infinite duration on a graph. Parity games provide a uniform framework for fixpoint-logic verification, model checking the modal $\mu$-calculus is equivalent to solving a parity game [32], and they are polynomial-time equivalent to tree-automata emptiness [50], which makes them suitable for application to ATL* model checking. Moreover, modern solvers achieve excellent performance in practice [29]. Although parity-game methods have been used in implementations of model-checkers for fragments of strategic logics (e.g. one-counter ATL* [74], SL[1G] [17]), and in LTL synthesis tools [54], no previous work has directly reduced full ATL* model checking to symbolic parity game solving. We address this gap by designing a three-phase symbolic construction that transforms a CGS $G$ and LTL path formula $\psi$ into a finite parity game whose winning strategies correspond to ATL* satisfaction.

1. **LTL→DPA.** Translate the LTL subformula $\psi$ into a deterministic parity word automaton (DPA).
2. **Product construction.** Synchronously compose the DPA with $G$ to obtain a parity game $\mathcal{G}_{G,\psi,A}$.
3. **Game solution.** Invoke an external parity-game solver to compute the winning region, and interpret the result to decide which states satisfy $\langle\langle A\rangle\rangle\psi$.

This algorithm is presented in a hybrid manner: we represent both the DPA and the CGS symbolically (using BDDs) and perform the product construction on these symbolic structures for efficiency, then export the resulting game explicitly to the external solver. After solving, we re-import the winning set as a symbolic BDD of satisfying states. Unlike the algorithm over Rabin automata, this single construction is sufficient. In the following, we briefly introduce parity-game basics and then present each phase in turn, including implementation notes and optimisations.

### 4.2.1 Parity Game Preliminaries

A min-parity game is a tuple

$$\mathcal{G} = (V,\, V_0,\, V_1,\, E,\, \Omega),$$

where $(V, E)$ is a finite directed graph, $V = V_0 \cup V_1$ partitions positions between Player 0 (Even) and Player 1 (Odd), and $\Omega\colon V \to \mathbb{N}$ assigns each vertex a *priority*. A play is an infinite path $v_0 v_1 v_2 \cdots$; Player 0 wins if the minimum priority occurring infinitely often is even, otherwise Player 1 wins. Max-parity games are defined similarly.

### 4.2.2 Deterministic Parity Automaton Construction

We begin by translating the LTL subformula $\psi$ into a DPA $\mathcal{A} = (S, \Sigma, \delta, q_0, c)$ again using the tool Rabinizer4 which implements the Safra-less construction from LTL to DPA through Limit-Deterministic Buchi Automata (LDBA) from [33]. Here $c\colon \delta \to \{0, \ldots, k-1\}$ assigns priorities to transitions, and thus is a transition-based acceptance DPA.

Since parity-game solvers require a state-based priority function $\Omega\colon S \to \{0, \ldots, k-1\}$, we convert $\mathcal{A}$ into an equivalent state-based DPA $\mathcal{A}'$ in two stages.

**Symbolic DPA Representation** We encode any state-based DPA $\mathcal{A}' = (S', \Sigma, \Delta, q'_0, \Omega)$ symbolically using BDDs over four variable vectors: $\vec{s}$ and $\vec{s'}$ for the current and next-state variables, $\vec{p}$ from the CGS for the propositional variables and $\vec{c} = (c_0, ..., c_{p-1})$, where $p = \lceil \log_2 k \rceil$ for the priority values. The transition relation is a BDD $\Delta(\vec{s}, \vec{p}, \vec{s'})$ and the priority map is a BDD $\Omega(\vec{s}, \vec{c})$.

**1. Spot's `autfilt` Lift** We first invoke Spot's `autfilt` tool [31] to convert $c$ into a state-based parity acceptance on an automaton in HOA format. When successful, the resulting DPA already carries $\Omega$ on states; we parse its states, transitions, and $\Omega$ into the symbolic structure above via our existing HOA parser.

**2. Priority-Tracking Fallback** If `autfilt` instead returns a Rabin or Büchi acceptance because it cannot encode certain transition-based parity conditions purely as state-based parity without changing the structure of the automaton, we develop a custom transformation based on the priority-tracking idea mentioned in [69]:

1. **Explicit parsing.** Read the HOA into an explicit list triples $(s, \sigma, s', c(s, \sigma, s'))$ representing the transitions. For any transition without a defined $c$-value, assign a dummy priority

$$p_{\max} = \begin{cases} k, & k \text{ even}, \\ k+1, & k \text{ odd}, \end{cases}$$

   so that $p_{\max}$ is the largest even number $\geq k$. This preserves the min-parity acceptance semantics.

2. **State-space expansion.** Let $k' = p_{\max} + 1$. Define

$$S' = \{(s, p) \mid s \in S,\ p \in [0..p_{\max}]\}, \quad \Omega(s, p) = p.$$

   Perform a breadth-first search from $(q_0, p_{\max})$: for each $(s, p)$ and each parsed $(s, \sigma, s', p')$, emit $\big((s, p), \sigma, (s', p')\big)$ and enqueue any unseen $(s', p')$. Encode each pair $(s, p)$ as the integer $s \cdot d + p$. Encode the transitions and priority function symbolically using the structure above.

In both cases, we obtain a fully symbolic, state-based DPA $\mathcal{A}_\psi$ with transition BDD $\Delta$ and priority BDD $\Omega$, ready for the parity-game product construction.

### 4.2.3 Parity Game Construction

We now build the turn-based parity game $\mathcal{G} = (V_0, V_1, E, \Omega)$ as defined in [34] for $\langle\langle A \rangle\rangle \psi$ by symbolic BDD operations on the CGS state variables $\vec{q}$, the DPA state variables $\vec{s}$, and the coalition-action variables $\vec{a}$.

Let $Reach(\vec{q}, \vec{s})$ be the BDD of reachable product states. Then

$$V_0(\vec{q}, \vec{s}) \ = \ Reach(\vec{q}, \vec{s}), \quad V_1(\vec{q}, \vec{s}, \vec{a}) \ = \ V_0(\vec{q}, \vec{s}) \ \wedge \ Act_A(\vec{q}, \vec{a}),$$

where $Act_A(\vec{s}, \vec{a})$ encodes exactly the coalition's joint moves available in CGS state $q$.

We construct two BDDs for the edge relation $E$:

1. $V_0 \to V_1$. Coalition picks an action, and that action is encoded into the next state.

$$E_{0\to1} = V_0(\vec{q},\vec{s}) \wedge \ \mathrm{Id}(\vec{q},\vec{q'}) \ \wedge \ \mathrm{Id}(\vec{s},\vec{s'}) \ \wedge \ Act_A(\vec{q'},\vec{a'}) \ ,$$

   where $\mathrm{Id}$ are identity BDDs linking current and next-state copies of $\vec{q},\vec{s}$, and we use the next-state version of the coalition joint actions BDD $Act_A$.

2. $V_1 \to V_0$. Edges from a $V_1$ vertex $(q,s,a)$ to a $V_0$ vertex $(q',s')$ exist exactly when there is some opponent move $a_{-A}$ such that the CGS evolves to $q'$ under $(a,a_{-A})$ and the DPA moves from $s$ to $s'$ on the labeling of $q$. This is encoded symbolically as

$$E_{V1\to V0} = V_1(\vec{q},\vec{s},\vec{a}) \wedge \delta(\vec{q},\vec{a},\vec{a_{-A}},\vec{q'}) \wedge \Delta(\vec{s},\vec{q},\vec{s'})$$

   where $\delta$ is the CGS transition BDD, and $\Delta$ the DPA transition BDD where again the transitions are encoded over the CGS current state variables rather than propositions. This transformation yields a fully symbolic parity game that can be exported and solved with a parity game solver, as described in the next section.

The game's priority map $\Omega\colon V \to \mathbb{N}$ is inherited from the DPA.

### 4.2.4 Parity Game Solving

We solve the explicit parity game using Oink [29], a high-performance C++ solver supporting many algorithms for parity solving such as Zielonka's recursion [79], small-progress measures, priority promotion, and quasi-polynomial-time tangle-learning (the one used) [28].

To export our symbolic game, we iterate over all satisfying assignments of the edge-relation BDD, decode each assignment into its CGS state $q$, DPA state $s$, and (for $V_1$) coalition action $a$, and write the corresponding vertex and edges in Oink's input format. We also extract the priority $p$ of each vertex and invert it via $p' = p_{\max} - p$ (since Oink expects a max-parity objective).

After Oink computes the winning player for every vertex, we re-import the result: we collect all $V_0$ vertices marked as winning for Player 0 into a BDD over $\vec{q},\vec{s}$. This final BDD precisely characterizes the set of states satisfying $\langle\langle A\rangle\rangle\psi$, completing the parity-game model-checking procedure.

### 4.2.5 Theoretical Complexity

Similarly to Section 4.1.2, we provide a concrete theoretical complexity for the implementation of ATL* model checking through parity games. Let G be a CGS of size $|G|$ and let $\psi$ be an LTL formula of length $|\psi|$. We have:

1. The LTL to DPA transformation for $\psi$ yields a DPA with transition-based acceptance and

$$N \ = \ 2^{2^{|\psi|}} \text{ states}, \quad K \ = \ 2^{|\psi|} \text{ priorities}.$$

2. Transforming the transition-based acceptance into state-based acceptance yields a DPA with in the worst-case through the priority tracking algorithm

$$N \ = \ 2^{2^{|\psi|}} \cdot 2^{|\psi|} \text{ states}, \quad K \ = \ 2^{|\psi|} + 1 \text{ priorities}.$$

3. The product parity game thus has (assuming the number of joint actions is negligible compared to the number of states)

$$n = |G| \times 2^{2^{|\psi|}} \times 2^{|\psi|} \text{ vertices, } K \text{ priorities.}$$

4. Solving a parity game with $n$ vertices and $k$ priorities can be done in quasi-polynomial time

$$O(n^{\log k})$$

Substituting $n = |G| \times 2^{2^{|\psi|}} \times 2^{|\psi|}$ and $k = 2^{|\psi|} + 1$ yields the following time complexity.

$$O((|G| \cdot 2^{2^{|\psi|}} \cdot 2^{|\psi|})^{\log(2^{|\psi|}+1)}) = O(|G|^{|\psi|} \cdot 2^{(2^{|\psi|}+|\psi|)\cdot|\psi|})$$

This shows that the algorithm is polynomial in $|G|$ with degree linear in $|\psi|$ and double exponential in $|\psi|$, which is the same overall 2-EXPTIME complexity but with different constants.

# Chapter 5

# Evaluation of Finite-Trace Model Checkers

This section presents an evaluation of the explicit and symbolic versions of the ATL$^*_f$ model checking algorithm developed in this project, focusing on two main aspects: correctness and performance.

Correctness was assessed by constructing a suite of small, hand-crafted CGS instances designed to exercise specific edge cases of the ATL$^*_f$ semantics. The expected outcomes for these instances were verified manually, and the tool's output was confirmed to match the theoretical results in all cases.

To assess performance and scalability, a synthetic benchmark was constructed, and two experiments were conducted: one evaluating scalability with respect to the size of the CGS, and another with respect to the size of the input formula.

All experiments in this chapter were run on the same hardware setup described in Section A.2.1. The following sections describe the synthetic benchmark developed, present the methodology used in each experiment, and analyse the results in terms of runtime, memory usage, and failure behaviour.

## 5.1 Benchmark Description

While temporal logics such as LTL and CTL have established benchmark suites, no such benchmark currently exists for ATL$^*$ model checking, particularly in the finite-trace setting. Thus, to evaluate performance and scalability, a custom benchmark suite was developed based on a scalable multi-agent counter system. This design was inspired by the single-counter scenario used in the LTL$_f$ synthesis benchmark suite [75], but adapted to a multi-agent setting suitable for strategic reasoning in ATL$^*_f$.

The system models two agents, A and B, each of which can either increment a shared counter (action `i`) or wait (action `w`). The global state is defined as a pair $(\texttt{curr\_count}, \texttt{step})$, representing the current value of the counter and the number of steps elapsed. The evolution is deterministic and synchronous: at each step, both agents choose an action, and the system updates the state accordingly. The system terminates once the number of steps reaches a maximum value $S$, and these states are treated as final states for the purpose of formula evaluation.

The benchmark is parameterised by two values: the maximum counter value $C$, and the maximum number of steps $S$. Together, these control the size and shape of the resulting CGS, enabling systematic scalability. The simplicity of the model makes it easy to generate automatically, while still being rich enough to support strategic temporal goals of increasing complexity.

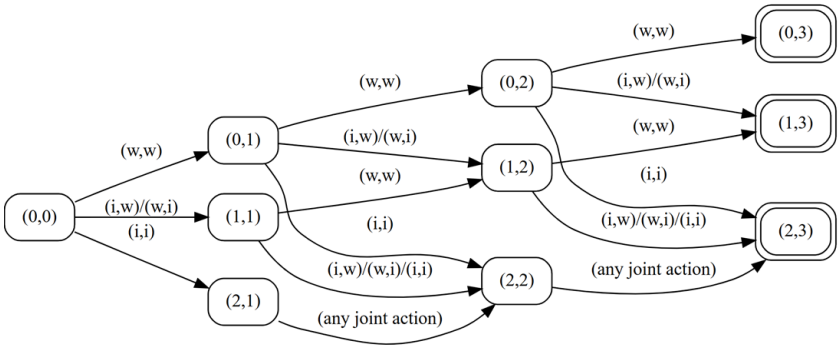


**Figure 5.1:** Example CGS instance from the benchmark with C=2,S=3

An example instance from the benchmark is shown in **Figure 5.1**, where each state is labelled with its $(\texttt{curr\_count}, \texttt{step})$ value, and transitions are labelled with the joint actions of the agents.

This benchmark serves as the basis for the two experiments described in the rest of this section: one varying the size of the CGS while keeping the formula fixed, and the other varying the formula complexity while holding the CGS fixed.

## 5.2 Experiment 1: Scaling the CGS

The goal of this experiment was to evaluate how the performance of the model checkers scales with the size of the underlying CGS, while keeping the temporal formula fixed. For this purpose, the benchmark generator was used to construct a series of counter systems by varying the parameters $C$ (maximum counter value) and $S$ (maximum number of steps). The ATL$^*_f$ formula used was:

$$\langle\langle A \rangle\rangle \, \mathsf{F} \, \texttt{counter\_max}$$

where `counter_max` is a proposition that holds in states where the counter reaches its maximum value. This formula remains fixed across all runs.

For each combination of $C$ and $S$, the model checker was run five times and the average runtime and peak memory usage were recorded. The results are presented as a function of state count.

### 5.2.1 Runtime Scaling

The runtime performance of the explicit and symbolic model checkers was evaluated as the number of CGS states increases. **Figure 5.2** presents the results for both implementations.

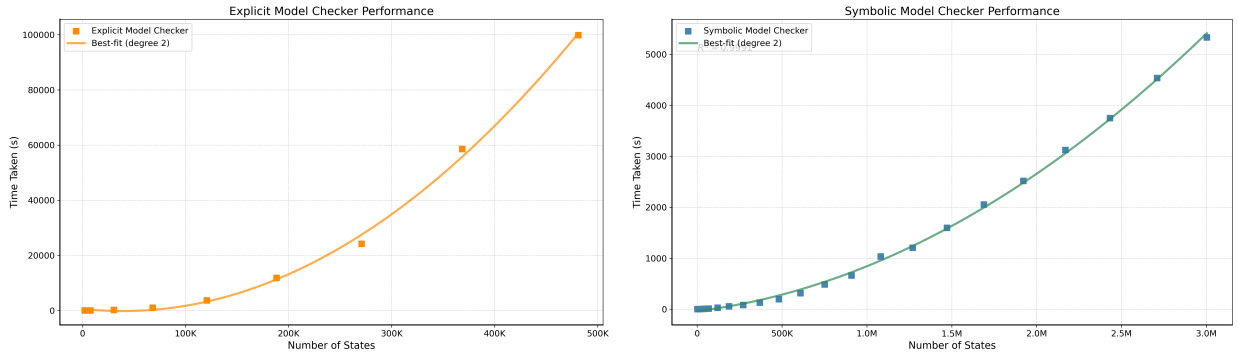


**Figure 5.2:** Runtime of the explicit (left) and symbolic (right) model checkers as the number of CGS states increases.

The explicit model checker exhibits a steep increase in runtime, with performance degrading rapidly beyond 100,000 states. This behaviour reflects the cost of constructing the full product state explicitly and performing repeated fixpoint computations over large sets. For the largest instance evaluated (nearly 500,000 states), the explicit model checker required over 100,000 seconds (27 hours) to complete.

In contrast, the symbolic model checker scales much more effectively. Despite also showing polynomial growth (approximately quadratic), the absolute runtimes remain far lower. The model with over 3 million states completes in under 5,500 seconds (approximately 1.5 hours), demonstrating far greater practical feasibility for verifying large systems. The improved performance is primarily due to the use of BDDs to encode the CGS and the product structure compactly, enabling efficient symbolic fixpoint operations without explicit state enumeration.

To highlight the comparative performance, **Figure 5.3** overlays the results of both implementations. As the plots show, the symbolic model checker begins outperforming the explicit implementation at relatively modest system sizes, and the gap widens dramatically for larger models. This demonstrates the clear scalability advantage of the symbolic approach. While both versions follow similar algorithmic logic, the underlying representation of the state space plays a crucial role in determining practical performance.

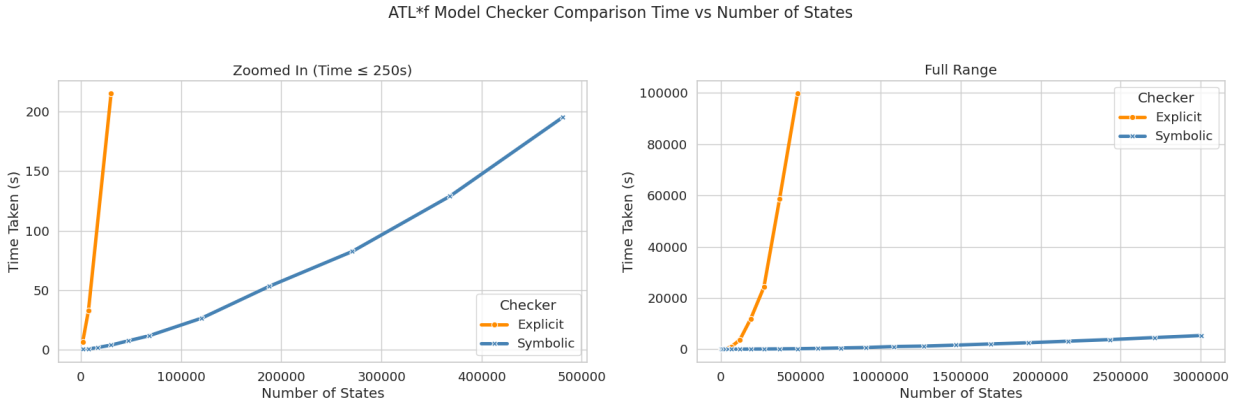


**Figure 5.3:** Comparison of runtime performance between the explicit and symbolic model checkers. Left: zoomed view (runtime $\leq$ 250s); right: full range.

### 5.2.2 Memory Usage

**Figure 5.4** shows the peak memory usage of the explicit and symbolic model checkers as the number of CGS states increases.

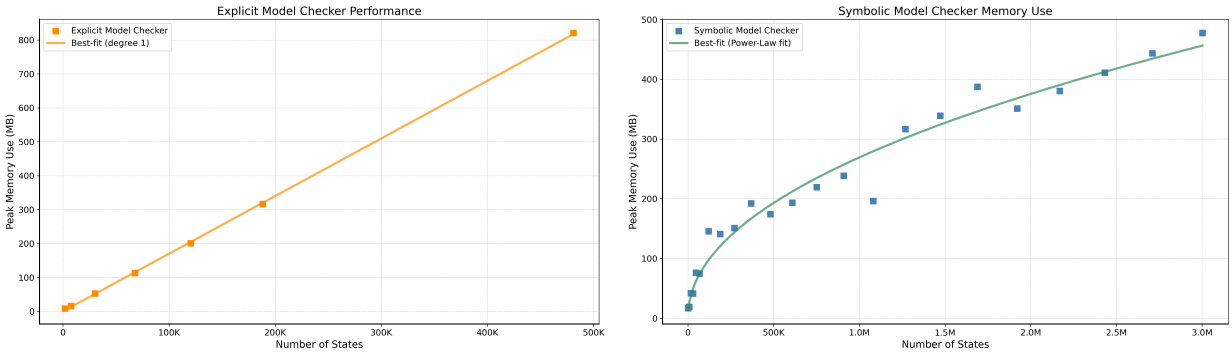


**Figure 5.4:** Peak memory usage of the explicit (left) and symbolic (right) model checkers as the number of CGS states increases.

The explicit model checker exhibits approximately linear memory growth, consistent with its use of `boost::dynamic_bitset` for storing state sets and an adjancency list representation for the product graph. For the largest instance, the peak memory consumption reaches about 830 MB. While the memory footprint is more manageable than the runtime at this scale, the high absolute cost and predictable growth suggest that state explosion remains a serious concern for even moderately large CGSs.

The divergence in time and space scaling for the explicit checker is notable. Although memory usage grows predictably with the size of the product graph, runtime increases far more sharply due to repeated fixpoint computations over explicit structures. This mismatch reflects the fact that memory costs are dominated by graph size, while computational costs are tied to how often those structures are traversed during verification.

In contrast, the symbolic model checker exhibits sublinear memory growth. The power-law trend observed in **Figure 5.4** (right) reflects the compactness of Binary Decision Diagrams (BDDs), which exploit shared substructures to represent large sets and transitions efficiently, resulting in more noticeable space savings as the underlying state space grows. For the largest tested instance (over 3 million states) the peak memory usage remains below 500 MB, significantly lower than that of the explicit model checker on much smaller instances.

This efficiency, however, is not purely tied to state count. As is common with BDD-based tools, memory consumption also depends on the structural regularity of the underlying system, the ordering of variables, and dynamic optimisations such as caching and reordering within the CUDD library. These factors lead to a more irregular growth pattern but enable much better scalability in practice.

Overall, the symbolic model checker achieves substantial memory savings compared to the explicit implementation. Together with its superior runtime performance, these results underline the practical advantages of symbolic representations for large-scale systems.

## 5.3 Experiment 2: Scaling the Formula

The second experiment investigates how the model checker's performance scales with increasing temporal formula complexity, while keeping the underlying CGS fixed. This complements the first experiment by isolating the cost associated with the construction and processing of the temporal objective itself, particularly the DFA synthesis stage and the subsequent product graph and fixpoint computation.

### 5.3.1 Experimental Setup

The CGS used for all runs in this experiment was a multi-agent counter system with parameters $C = 40$ and $S = 35$, resulting in a graph with approximately 1,000 states. This size was chosen to keep the underlying state space small enough to avoid timeouts or memory exhaustion from the CGS alone, ensuring that the formula size was the primary variable affecting performance.

The temporal formula used was a nested conjunction, each conjunct targeting a specific counter value. Formally, the formula has the shape:

$$\langle\langle A, B\rangle\rangle \, \mathsf{F}\, p_1 \wedge \mathsf{X}(\mathsf{F}\, p_2 \wedge \mathsf{X}(\mathsf{F}\, p_3 \wedge \ldots \wedge \mathsf{X}(\mathsf{F} p_n)))$$

where each proposition $p_i$ holds at states where the counter equals $i$. The depth of nesting ($n$) was varied from 1 to 20, increasing the size of the formula and the resulting complexity of DFA. Each instance was again run 5 times, and average runtime was recorded. Memory usage was not included in this experiment, as the dominant memory cost stems from the DFA construction phase, which is delegated to external tools and thus not captured in internal profiling data. Furthermore, the tool (Lydia, Lisa2 or LTLF2DFA) used to construct the DFA was tracked for each formula size.

### 5.3.2 Results

**Figure 5.5** shows the average runtime for formulas of size $1 \leq n \leq 20$, including the tool used for DFA construction and lines of best fit for both implementations. Both implementations exhibit similar scaling behaviour. For small to moderately sized formulas ($n \leq 11$), runtime grows approximately linearly. However, the symbolic model checker consistently achieves lower runtime in this range, with a noticeably flatter slope. This reflects the efficiency gains of symbolic product construction and fixpoint solving, which dominate the verification cost when the LTL formula remains relatively small.

From $n = 12$ onwards, an exponential growth trend emerges in both implementations. This shift is attributed to the rising cost of DFA construction, which becomes the dominant factor in total runtime. The increasing size and complexity of the generated automata not only affect translation time but also slow down the subsequent model checking phases. Nevertheless, the symbolic checker continues to outperform the explicit one throughout the range, maintaining a practical runtime even for the largest successful instance ($n = 18$).

The performance of LTLF2DFA in this range exhibits some erratic behaviour: the runtime for $n = 15$ is unexpectedly lower than that of Lydia at $n = 14$, which reflects trends already observed in the evaluation of LTLf-to-DFA tools in Appendix A. Specifically, LTLF2DFA has previously shown lower translation times for certain larger formulas despite struggling with smaller ones.

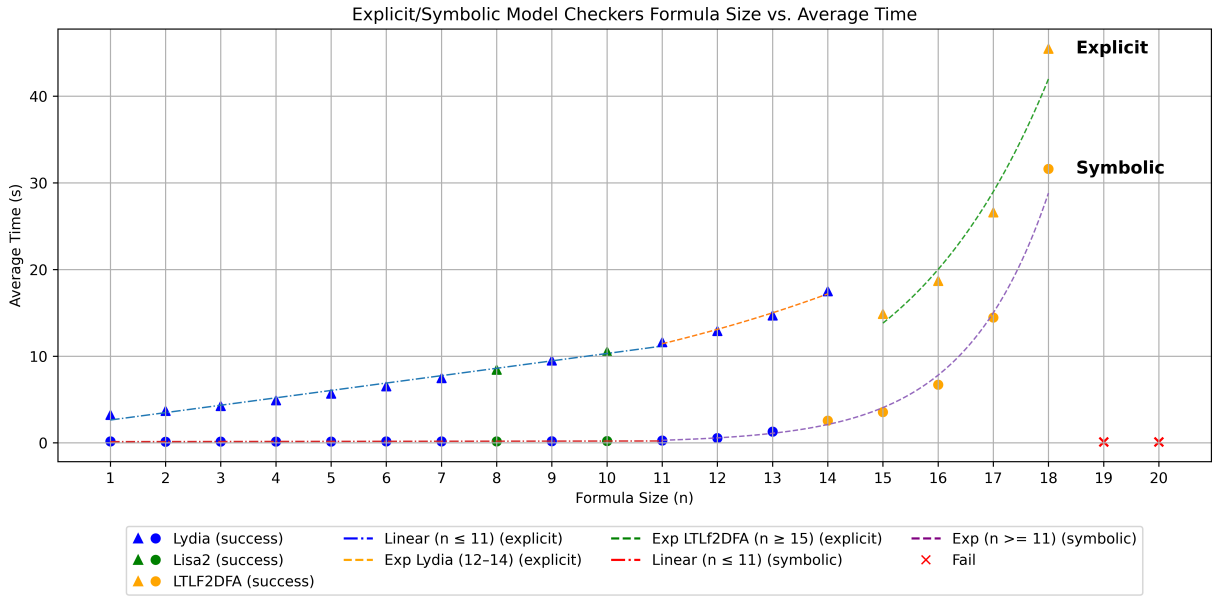


**Figure 5.5:** Runtime of the explicit and symbolic model checkers as formula size increases. Marker colour indicates the tool used for DFA generation. Dashed lines show fitted curves for linear and exponential trends.

For $n \geq 19$, all tools fail to construct the DFA within resource limits. This reflects the sharp exponential growth in automaton size and translation time, making verification infeasible beyond this point. Notably, the total runtime never exceeds 50 seconds for any successful instance, including those with larger formulas. This suggests that the verification cost after DFA construction remains efficient, likely due to the relatively small size of the CGS. The failure for $n \geq 19$ is therefore not attributable to excessive runtime, but rather to memory exhaustion during the DFA synthesis phase.

The distribution of selected tools across formula sizes also reinforces the decision to use a parallelised translation strategy. Lydia and Lisa2 are preferred at smaller sizes, consistent with their superior performance on structured formulas in Chapter A, while LTLF2DFA is better at larger formulas. Without this hybrid strategy, the model checker would have failed at $n = 17$; thus by leveraging the strengths of all three tools, the system remains faster and more robust across the full range of inputs.

In summary, Experiment 2 confirms the symbolic model checker's advantage in both efficiency and scalability, particularly for small to moderately complex formulas. While both implementations are ultimately bounded by the limitations of the external DFA translation phase, the symbolic backend remains consistently faster and better suited for realistic specifications.

## 5.4 Evaluation Summary

The evaluation results highlight the scalability advantages of the symbolic model checker across different dimensions of the verification problem. In Experiment 1, which varied the size of the CGS, the symbolic implementation consistently and significantly outperformed the explicit version in both runtime and memory usage. This demonstrates the effectiveness of symbolic techniques in

mitigating state explosion, allowing verification to scale to models with millions of states.

In Experiment 2, which varied the size of the temporal formula, both model checkers encountered limitations due to the external DFA synthesis step. However, the symbolic checker remained more efficient across the range of successful instances, particularly for smaller and moderately sized formulas that are most common in practice. The use of a parallel tool strategy for DFA generation proved to be an effective design choice, combining the strengths of different synthesis tools to improve both performance and robustness.

Together, these experiments confirm that symbolic model checking offers practical benefits in performance, scalability, and robustness. The results also motivate further evaluation in domain-specific settings. The next section explores this in the context of a cybersecurity training ground scenario, where the symbolic model checker is applied to a real-world-inspired system to verify complex strategic properties.

# Chapter 6

# Modelling of a Multi-Agent Cybersecurity Defense Scenario

In this chapter, we present the formal modelling of a realistic cybersecurity scenario in which multiple defensive agents cooperate to detect and mitigate an intelligent attacker. Beyond benchmarking the scalability of the symbolic ATL$^*_f$ model checker in a complex, real-world setting, this chapter also aims to demonstrate the practical benefits of formal verification for strategic defense planning. The key objectives of this modelling are to determine the minimum budget required by the defenders to guarantee absolute protection against the attacker, ensuring that no server can be compromised under any possible attack strategy and to identify the most cost-effective defense strategy among several heuristics. We base our scenario on the Multi-Agent Reinforcement Learning (MARL) cybersecurity training ground introduced by Wiebe et al. [77], representing the core interactions in a CGS and abstracting low-level details. This provides a suitable and structured environment for verifying ATL$^*_f$ properties related to defense guarantees while serving as a benchmark for assessing the performance of the symbolic model checker under realistic conditions.

## 6.1 Model Description

### 6.1.1 Scenario Overview

We consider a protected network of five interconnected servers, monitored by two cooperative defender agents and contested by a single intelligent attacker. In practical terms, each server represents a critical service or host within an enterprise infrastructure, defenders model intelligent Intrusion Prevention Systems (IPSs) with limited time and budget, and the attacker represents an Advanced Persistent Threat (APT) seeking to achieve one of three classic objectives:

1. **Confidentiality Breach**: gain administrator-level access to exfiltrate sensitive data.
2. **Integrity Violation**: insert or modify files to corrupt system behaviour.
3. **Availability Attack**: deploy denial-of-service malware to render a host unusable.

Defenders may monitor individual servers-analogous to querying host-based sensors-to observe a fixed set of six Boolean flags, detailed in Section 6.1.2. Based on these observations and a finite budget, they can remove malicious processes or fully restore a server image. The attacker, in turn, can scan for vulnerabilities, exploit or escalate privileges, and, depending on its goal, tamper with files or install a DoS-style payload.

This abstraction captures key trade-offs in real deployments: limited visibility, resource constraints, and the need for timely reaction. In the following sections we show how these elements are encoded in a Concurrent Game Structure suitable for ATL$^*_f$ verification.

### 6.1.2 State Space and Observables

The following state components describe the *availability* scenario; the *confidentiality* and *integrity* scenarios use a similar structure, with only the goal-specific variables (e.g. tamper flags, data-repair actions) differing.

The system is modelled as a global state $s \in S$ comprising the following components:

- **Attacker position** $p \in \{0, \ldots, 4\}$: the server on which the attacker currently resides.
- **Privilege levels** priv$[i] \in \{0, 1, 2\}$ for each server $i$: 0 = no access, 1 = user, 2 = admin.
- **Vulnerability map** scanned$[i] \in \{0, 1\}$: whether server $i$ has been scanned.
- **Availability map** down$[i] \in \{0, 1\}$: whether server $i$ is unavailable due to a DoS payload.
- **Defender budgets** $B$: remaining combined budget for defenders $D_1$ and $D_2$.
- **Suspicion buckets** $\sigma_{j,i} \in \{0, 1, 2\}$: current suspicion level that defender $D_j$ assigns to server $i$.
- **Step counter** $t$: the current time step, $0 \leq t \leq T$.
- **Server flags** $F_i = (F_{i,1}, \ldots, F_{i,6})$: six Boolean indicators per server $i$, defined below.

In a real-world defense system, defenders do not have direct access to most of these components; instead, they may invoke a `monitor` action on server $i$ to observe its current flag vector $F_i$. The six flags, which abstract common intrusion-detection and host-monitoring alerts, are:

$F_{i,1}$ : "server $i$ was scanned last turn",
$F_{i,2}$ : "exploit ran on $i$ last turn",
$F_{i,3}$ : "no attacker session on $i$",
$F_{i,4}$ : "$i$ is currently unavailable",
$F_{i,5}$ : "user-level process running on $i$",
$F_{i,6}$ : "admin-level process running on $i$".

These flags are part of the global state but are only revealed to a defender when they choose to monitor the corresponding server. All other variables remain hidden until inferred via subsequent actions and observations.

### 6.1.3 Agent Action Sets

At each step, each agent selects one action from its individual set. Each defender action has a cost, associated with it's level of disruption on the system. Logical preconditions such as requiring scanned$[i] = 1$ before `exploit` or sufficient suspicion before `restore` are enforced by the protocol function (Section 6.1.4).

**Defender actions**

- `monitor`($i$): query server $i$ to observe its flag vector $F_i$.
- `remove`($i$): evict any user-level malicious process on $i$.
- `restore`($i$): re-image $i$, clearing all sessions and processes.
- `do_nothing`: skip action.

In the *integrity* scenario, defenders also have an `analyze`($i$) action to inspect file-system metadata to detect tampering and a `data_repair`($i$) action to remove or revert modified files on $i$.

**Attacker actions**

- `scan`($i$): probe $i$ for vulnerabilities.
- `exploit`($i$): leverage a known vulnerability to obtain user-level access.
- `escalate`($i$): promote from user to admin privileges on $i$.
- `do_nothing`: skip action.

Additionally in the *integrity* scenario: `tamper`($i$), modifying or corrupting files (requires appropriate privileges) and in the *availability* scenario: `deny`($i$), install a DoS-style payload to render $i$ unavailable.

### 6.1.4 Abstraction of Imperfect Information

By default, $\mathrm{ATL}^*_f$ model checking operates over perfect-information Concurrent Game Structures, where all agents observe the full global state. In our cybersecurity scenario, granting defenders this level of visibility would allow unrealistic reactions—for example, always removing the attacker from the exact node. Unfortunately, the model checking problem for ATL$^*$ under imperfect-information semantics is undecidable [8], so it is necessary to simulate this restricted information within the perfect information model.

To approximate partial observability while retaining a decidable perfect-information framework, we use a dynamic action-restriction mechanism via the protocol function. Each defender $D_j$ maintains per-server suspicion buckets $\sigma_{j,i}$ (Section 6.1.5), computed only from the flags it has actually observed. The protocol function then permits high-impact actions (e.g. `remove`, `restore`) on server $i$ only if $\sigma_{j,i}$ meets the required threshold. In this way, we capture realistic uncertainty: defenders cannot act on an attacker's true position until sufficient evidence has been gathered, yet remain within a perfect-information CGS suitable for symbolic $\mathrm{ATL}^*_f$ verification.

### 6.1.5 Suspicion Heuristics

To simulate realistic uncertainty under perfect-information model checking, each defender $D_j$ assigns a suspicion bucket $\sigma_{j,i} \in \{0, 1, 2\}$ to each server $i$ based only on the flags $F_i$ it has observed via `monitor` actions. The bucket determines which defensive actions $D_j$ may legally invoke on $i$. The following are the decision heuristics for the *availability* scenario; other scenarios follow a similar structure, although the *integrity* scenario, for instance, uses four buckets to account for two additional defender actions.

$\sigma_{j,i} = 0 \implies \{\texttt{monitor}\}, \quad \sigma_{j,i} = 1 \implies \{\texttt{monitor}, \texttt{remove}\}, \quad \sigma_{j,i} = 2 \implies \{\texttt{monitor}, \texttt{remove}, \texttt{restore}\}.$

We implement four heuristic strategies for computing $\sigma_{j,i}$:

1. **Conservative Heuristic.** Each flag $F_{i,k} \in \{0, 1\}$ carries a danger weight $w_k$ (e.g. higher for exploit or admin-process flags). Defender $D_j$ computes the weighted sum

$$S_i = \sum_{k=1}^{6} w_k F_{i,k},$$

and applies thresholds $0 \leq T_1 < T_2$ so that $\sigma_{j,i} = 0$ if $S_i < T_1$, $\sigma_{j,i} = 1$ if $T_1 \leq S_i < T_2$, and $\sigma_{j,i} = 2$ otherwise.

2. **Aggressive Heuristic.** Inspired by modern zero-trust architectures [66], any critical flag, specifically, $F_{i,2}$ ("exploit ran") or $F_{i,6}$ ("admin-level process"), immediately elevates $\sigma_{j,i}$ to 2. Otherwise if $F_{i,4}$ ("scanned") or $F_{i,1}$ "user-level process" is set we set the suspicion level to 1, and 0 otherwise.

3. **Proportional Heuristic.** We compute the weighted sum as for the conservative heuristics, and divide this by the maximum possible danger weight to get a normalized danger score. Thresholds are then applied to this ratio to set the suspicion levels. This is inspired by risk-matrices approaches where the severiy scores are normalized into percentiles.

4. **Diversity Heuristic**. We categorise each flag by its type (i.e. recon, exploit, user, none), and set the suspicion level based on the number of different types seen. This is inspired by the MITRE cyber kill-chain description, where an attack is divided into phases [55].

Each heuristic reflects different operational philosophies (risk-weighted analysis, aggressive zero-trust or conservative postures, stage-based detection) and allows us to compare their impact on verified defender guarantees.

## 6.2 Evaluation on Symbolic Model Checker

An evaluation was conducted on the symbolic model checker by increasing the number of steps $n$ and the defender budget $b$ to increase the total number of reachable states (i.e. number of reachable combination of the variables defined in Section 6.1.2) and assess its practical scalability. The ATL$^*_f$ formula verified is $\langle\langle$Attacker$\rangle\rangle$FG compromised_1 The evaluation was conducted over the three attacker goals (confidentiality, availability and integrity). Results are presented in **Figure 6.1**.


**Figure 6.1:** Model Checking Runtime as number of states increases for each attacker-goal scenario.

The symbolic model checker shows strong scalability in this complex, real-world model. For models with over one million states, verification remains under an hour, demonstrating that it can handle large-scale inputs efficiently. Runtime growth is consistent across the three scenarios, but lower overall in the availability setting. This is likely due to a more constrained action space: defenders' choices depend on both the current suspicion bucket and previously executed actions (e.g. a file must be identified before it can be removed), reducing the branching factor and simplifying symbolic operations.

In contrast, the confidentiality scenario permits a wider range of actions at each state due to its more limited suspicion partitioning, resulting in a larger joint action space and increased runtime. These results confirm that both the size of the reachable state space and the branching factor significantly influence performance.

## 6.3 Verification of Properties

To also give insights into the kind of useful strategic ATL* properties that can be verified within the model, we propose to find the minimum budget needed for the defenders to be able to prevent the compromise of 1 or more servers in 10 steps in the confidentiality scenario. This property can be expressed in ATL* as

$$\langle\langle D_1, D_2 \rangle\rangle \mathsf{FG} \neg \text{compromised\_1}$$

which expresses that the coalition $D_1, D_2$ can eventually ensure that no server remains compromised.

We perform a binary search over budget values in the range $b \in [1..20]$ to find the minimum value for which the model checker returns true for the formula above. This is repeated for each of the four heuristics used by the defenders to determine the suspicion score as defined in Section 6.1.5.

| Heuristic | Minimum Budget |
|---|---|
| Proportional | 10 |
| Aggressive | 12 |
| Conservative | 14 |
| Diversity | 14 |

**Table 6.1:** Minimum budget required for each heuristic to satisfy the formula.

The results show that the most cost-effective heuristic in this confidentiality scenario is the proportional strategy. This heuristic is effective since it can identify all kinds of threats and escalates suspicion quickly. The aggressive strategy performs slightly worse, likely due to its narrower detection strategy. The conservative and diversity-based are not very effective, since they are too slow at escalating suspicion levels which reduces the possibilities of intervention before compromise.

These results show the practical benefits of ATL* for verification, demonstrating how it can provide absolute formal guarantees regarding properties which in a broader context can be relevant to MARL agent training and can provide safe operational thresholds on the training ground being modelled.

# Chapter 7

# Evaluation of Infinite-Trace Model Checkers

This chapter presents an experimental evaluation of the two infinite-trace ATL$^*$ model checkers developed in this work: the Rabin tree automata–based approach and the parity game–based approach. The evaluation focuses on assessing their performance on a synthetic benchmark based on the multi-agent counter model introduced in the finite-trace setting, adapted here for infinite traces. For this benchmark, we conduct two experiments: one measuring scalability with respect to the size of the concurrent game structure (CGS), and another measuring scalability with respect to the size of the ATL$^*$ formula.

Additionally, to contextualise the performance of the parity-based model checker, we compare it against two relevant baselines: the finite-trace model checkers from Chapter 5, using their respective counter benchmarks, and the SL[1G] module of MCMAS on a family of fair scheduler synthesis models, which are naturally suited to infinite traces.

All performance measurements are based on wall-clock execution time, and all model checkers were run under identical hardware conditions as described in Appendix A. The following section describes the infinite-trace counter benchmark used throughout these experiments.

## 7.1 Infinite-Trace Counter Benchmark

The benchmark used to evaluate the infinite-trace model checkers is based on a modulo-$m$ counter, extended from the finite-trace version introduced earlier. It is parameterised by the number of agents $n$ and the maximum counter value $m$, and is designed to be both simple and scalable.

Each state encodes a single counter value $s \in \{0, \ldots, m-1\}$, and agents jointly update the counter by selecting actions from $\{\mathsf{inc}, \mathsf{wait}\}$. At each step, the next counter value is given by:

$$s' = (s + \text{number of } \mathsf{inc} \text{ actions}) \bmod m.$$

This results in a CGS with $m$ states and a branching factor of $2^n$, making it suitable for controlled experiments on both structural and formula complexity. Each state is labelled with a unique proposition $p_s$, allowing ATL$^*$ to refer directly to counter values. **Figure 7.1** illustrates an example instance with $n = 3$ agents and $m = 4$ counter values.

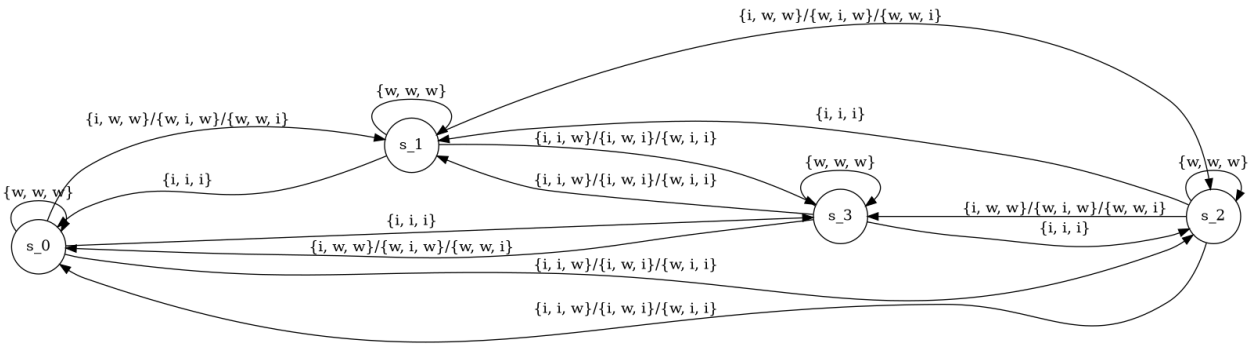


**Figure 7.1:** Example CGS with $n = 3$ agents and counter values modulo $m = 4$.

## 7.2 Experiment 1: Scaling the CGS

This experiment evaluates how the two infinite-trace ATL* model checkers scale with respect to the size of the concurrent game structure (CGS). We consider two orthogonal ways to grow the CGS:

- **Experiment 1a:** Increase the number of states by varying the counter modulus $m$.
- **Experiment 1b:** Increase the branching factor by varying the number of agents $n$.

In both cases, the verified formula is fixed as:

$$\langle\langle A_1 \rangle\rangle \, \mathbf{GF} \, p_{m-1},$$

which expresses that the coalition can ensure that the counter reaches its maximum value infinitely often.

We now present the results for each subexperiment independently.

### 7.2.1 Experiment 1a: Increasing the Number of States

**Figure 7.2** reports the runtime of both model checkers as the value of $m$ increases while keeping $n = 2$ fixed.

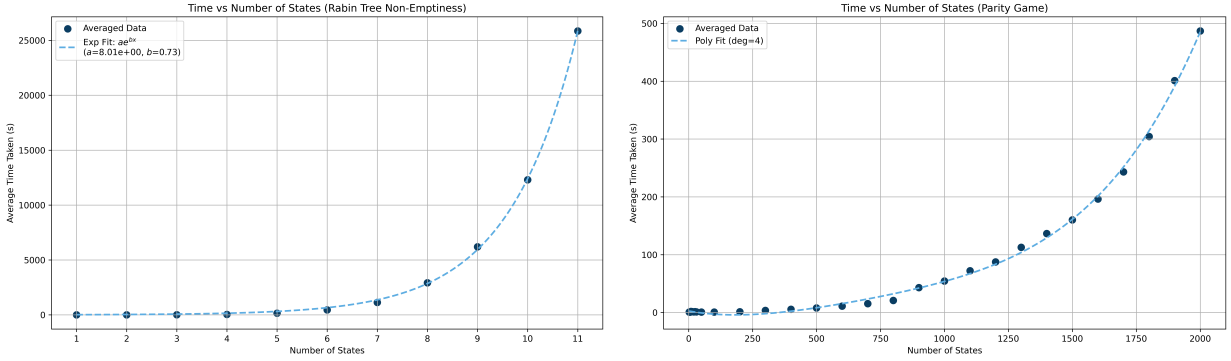


**Figure 7.2:** Runtime of the Rabin-based (left) and parity-game–based (right) ATL* model checkers as CGS size increases via the counter modulus $m$.

The Rabin-based checker scales poorly: models with more than 10 states already lead to prohibitively long runtimes (e.g., nearly 7 hours for $m = 11$). This bottleneck originates in the non-emptiness algorithm, where automata transformations introduce exponential blow-up in the number of states and the complexity of the transition function. The BTA, DRTA, and symbolic product construction are efficient at this scale, but the cost of operations such as co-WRBAA to co-WRAA conversion and simple-WAA construction becomes dominant, as illustrated in **Figure 7.3**.

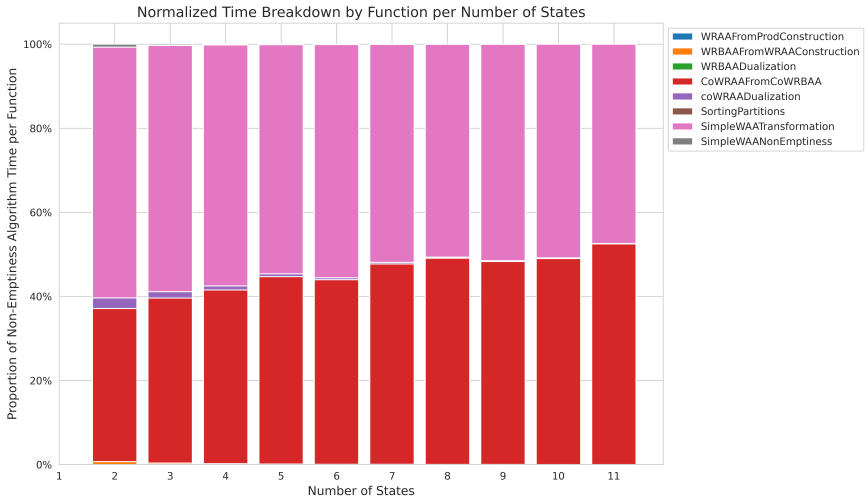


**Figure 7.3:** Normalised time spent per function of the Rabin non-emptiness algorithm.

This unfortunately renders the Rabin-based model checker unsuitable for practical verification of ATL* specifications, at least in its current form. By contrast, the parity-game approach exhibits much better scalability, solving models with thousands of states in under 10 minutes. This is likely due to three key factors: the efficient symbolic construction of the parity game, the use of highly optimised external solvers (in this case, Oink) and the fact that the algorithm only needs to be applied once per strategic subformula, rather than once per state. **Figure 7.4** further highlights the performance gap between both approaches.

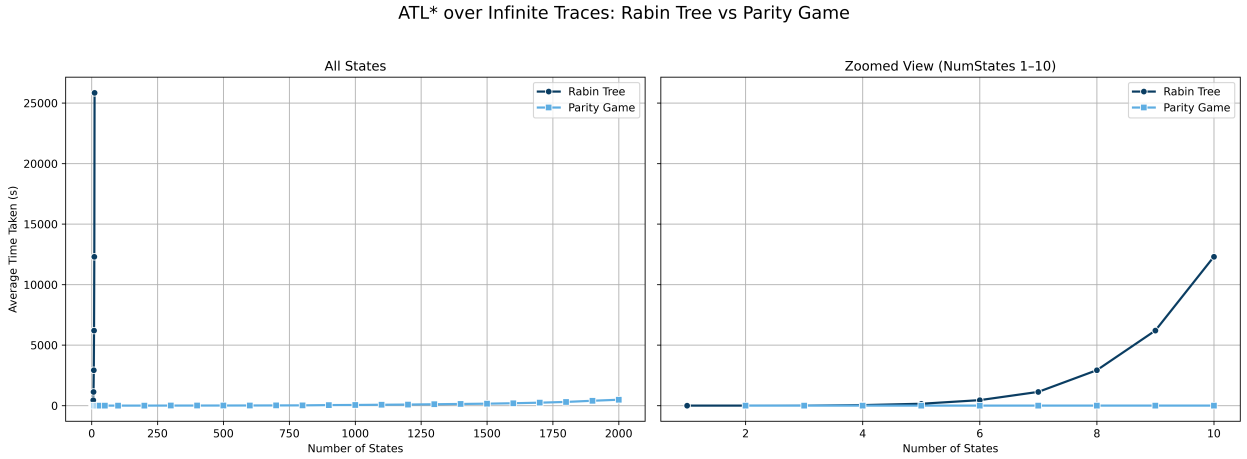


**Figure 7.4:** Comparison between Rabin and parity approaches. Left: full range of counter values tested. Right: zoom on $m \leq 10$.

This difference can be partly attributed to the theoretical complexity bounds of the two approaches (see Sections 4.1.2 and 4.2.5). Although both are 2EXPTIME-complete in the worst case, the structure of the complexity expressions differs substantially. In the Rabin approach, the size of the CGS appears in the base of an exponential term dependent on the formula size. In contrast, in the parity-game approach, the size of the CGS appears only as a multiplicative factor whose polynomial degree depends linearly on the formula size. This structural difference, along with the factorial term in the Rabin case, both due to better complexity of solving parity games compared to NRTA non-emptiness, leads to significant performance divergence in practice. Moreover, the parity-game solution benefits from decades of algorithmic and engineering optimisations, whereas the Rabin pipeline uses a custom, less mature implementation.

### 7.2.2 Experiment 1b: Increasing the Number of Agents

In this experiment, we evaluate how performance scales with the number of agents by increasing both $n$ (agents) and $m$ (maximum counter value), which together determine the size and branching factor of the CGS. Higher agent counts result in larger joint action spaces and more complex successor sets per coalition action. This experiment is performed only on the parity-game–based model checker, as the Rabin-based version does not scale to models of this size. The results are shown in **Figure 7.5**.

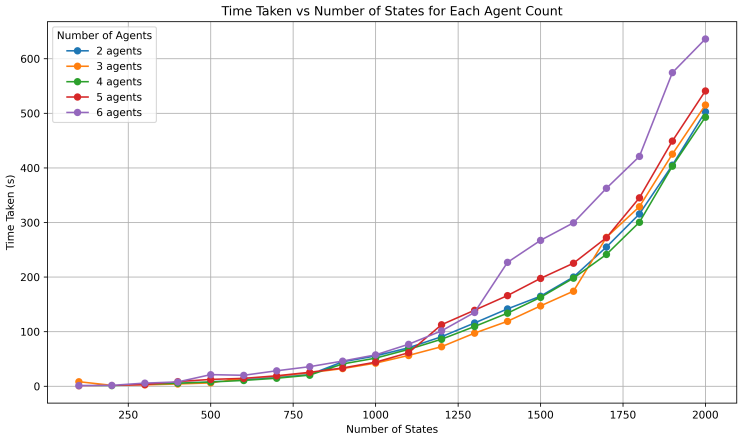


**Figure 7.5:** Runtime vs. CGS size for varying numbers of agents.

The results indicate that although runtime increases slightly with the number of agents, particularly in the $n = 6$ case, the overall scaling behaviour remains consistent. The runtime curves follow similar polynomial trends across agent counts, suggesting that the symbolic construction handles increases in joint action space and branching factor efficiently. This reinforces the scalability of the parity-game–based model checker with respect to both state space size and action space complexity.

## 7.3 Experiment 2: Scaling the Formula

This experiment follows the methodology of the finite-trace counterpart presented in Section 5.3, evaluating scalability with respect to formula size under a fixed CGS. The model consists of $n = 2$ agents and $m = 100$ counter values, resulting in a CGS with 100 states. The ATL$^*$ formulas are

constructed to increase in nested depth from $n = 1$ to $n = 20$, following the pattern:

$$\langle\langle A_1, A_2 \rangle\rangle \, \mathbf{GF}\, p_1 \wedge \mathbf{X}(\mathbf{GF}\, p_2 \wedge \mathbf{X}(\mathbf{GF}\, p_3 \wedge \cdots \wedge \mathbf{X}(\mathbf{GF}\, p_n)))$$

Due to the poor scalability demonstrated in Experiment 1a, the Rabin-based model checker is omitted from this experiment. We evaluate only the parity-based model checker. The results are shown in **Figure 7.6**.

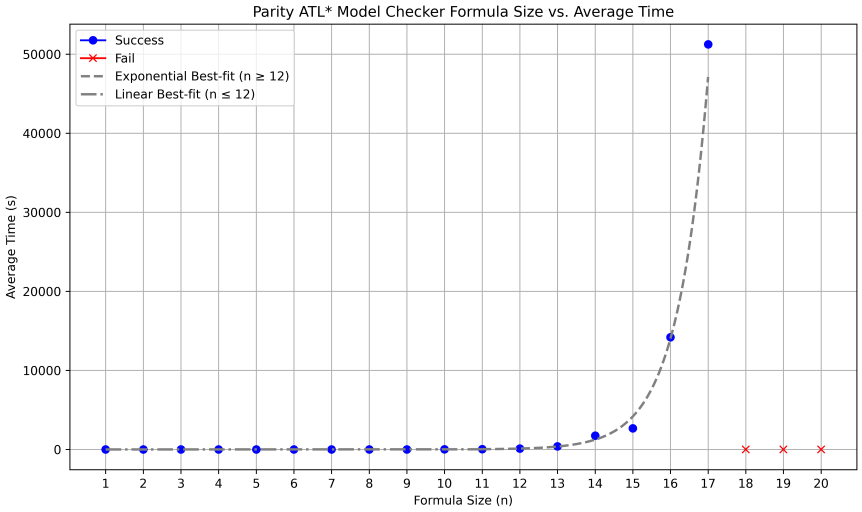


**Figure 7.6:** Runtime of the parity-based model checker as formula size increases. Failures due to memory exhaustion are marked.

The model checker performs efficiently for small formulas, with runtime increasing roughly linearly up to $n = 12$. This scaling behaviour implies that in this region the main computational effort lies in symbolic product construction and parity game solving, rather than in automaton generation.

Beyond this point, however, runtime grows exponentially. For instance, verifying the formula with $n = 17$ takes over 14 hours. This steep growth is primarily due to the LTL-to-DPA translation step, which begins to approach its worst-case double exponential theoretical complexity. The size of the generated DPA increases rapidly with formula depth, and the subsequent conversion from transition-based to state-based acceptance also becomes a significant bottleneck. Additionally, the `ltl2dpa` tool (Rabinizer4) fails with out-of-memory errors for $n > 17$, indicating that the construction of DPAs is not only significantly slower but also slightly less robust than analogous finite-trace translations to DFAs.

Overall, while the parity-based approach handles moderately complex formulas well, it faces significant limitations when faced with deeply nested temporal specifications.

## 7.4 Comparison with Finite-Trace Model Checkers

In this section, we compare the performance of the infinite-trace ATL* model checker based on parity games with the two finite-trace model checkers developed in Chapter 3, using explicit-state and symbolic representations. The goal of this comparison is to quantify the practical impact of trace semantics on performance.

To ensure a fair evaluation, we replicate the finite-trace experiments from Section 5.3 using a CGS with 100 states (achieved by setting $c = 20$, $s = 9$), matching the model size used for the infinite-trace experiments. We evaluate all three implementations on both model-size scaling (Experiment 1) and formula-size scaling (Experiment 2). The results are shown in **Figure 7.7**, using a logarithmic scale to accommodate the wide range of runtimes observed.

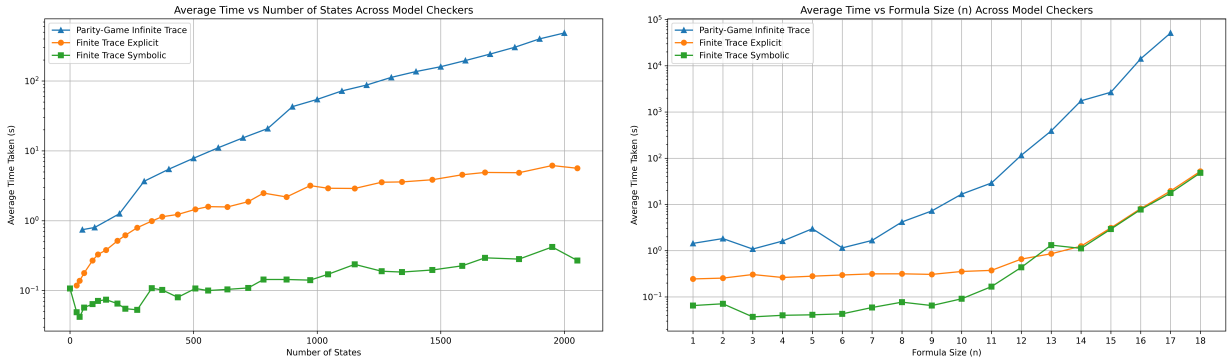


**Figure 7.7:** Comparison of finite-trace (explicit and symbolic) and infinite-trace (parity-game–based) model checkers. Left: runtime vs. CGS size. Right: runtime vs. formula size. Log scale used.

The results confirm that finite-trace model checkers offer substantial performance benefits across both axes of scalability. Both the symbolic and explicit finite-trace implementations outperform the parity-based infinite-trace checker by several orders of magnitude. The gap is especially pronounced in model-size scaling, where the parity-based checker encounters increased overhead from parity game solving. In the formula-scaling experiment, although all approaches eventually exhibit exponential growth, the infinite-trace version reaches much higher runtimes more quickly, scaling significantly worse.

Importantly, this does not indicate a failure of the infinite-trace approach, but rather reflects the inherent complexity of reasoning over infinite behaviors. The symbolic parity-based checker remains a viable and correct method for ATL* verification, particularly in domains where infinite-trace properties are required and no finite abstraction is appropriate. Indeed, as shown in the next section, it compares favourably to existing tools that handle more expressive logics.

Although both model-checking problems are 2EXPTIME-complete in theory, these results again underscore how theoretical complexity classes often conceal significant practical differences. The finite-trace algorithm incurs only linear dependence on the size of the model, and the doubly exponential term in the formula size is comparatively lightweight in both its structure and in the actual size of the generated automata. In contrast, the parity-based infinite-trace checker incurs a polynomial dependence on model size with degree linear in the size of the formula and suffers from more complex and less predictable growth in automata size. In practice, $\text{LTL}_f$-to-DFA conversions often yield automata far smaller than their worst-case bounds, while the corresponding $\omega$-automata for LTL can remain large even when generated with optimized tools like Rabinizer4.

Thus, these results confirm that, when a finite-trace abstraction is applicable to the scenario being modelled, it should be strongly preferred due to the substantial performance advantages it provides. However, the parity-based checker remains an effective and scalable option within its

domain, offering a principled and automata-theoretic solution to full ATL* model checking over infinite traces.

## 7.5 Comparison with SL[1G] Model Checker

While no existing tools implement model checking specifically for ATL*, there are tools for more expressive logics that subsume it. One such logic is the one-goal fragment of Strategy Logic (SL[1G]), whose semantics strictly generalise ATL* and whose model checking problem is also 2EXPTIME-complete [57]. This makes SL[1G] a compelling basis for indirect ATL* verification, especially when paired with an efficient implementation.

In this section, we compare our parity-game–based ATL* model checker against the SL[1G] model checker implemented as an extension to MCMAS [16], using a common benchmark. This comparison is motivated by both tools targeting strategic reasoning under perfect information and full recall using symbolic parity game constructions over LTL path formulas. However, there are important differences. SL[1G] supports nested quantification over strategies and agent bindings, while ATL* restricts quantification to agent coalitions.

The SL[1G] model checker in MCMAS is fully symbolic and based on the construction of deterministic parity automata (DPAs) for the temporal components of the formula. These are then combined with a symbolic representation of the interpreted system to build a parity game, which is solved using a symbolic implementation of Zielonka's algorithm [16].

### 7.5.1 Fair Scheduler Synthesis Benchmark

To evaluate both model checkers, we use the fair scheduler synthesis problem which is a standard benchmark for strategic reasoning over infinite traces. The system consists of $n$ processes (agents $P_1, \ldots, P_n$) and a dedicated Scheduler agent, which arbitrates access to a shared resource. The Scheduler must ensure mutual exclusion (only one process may access the resource at a time) and fairness (no process waits indefinitely).

The global state includes:

- **Process states:** Each process is either `in` (idle), `wt` (waiting), or `rs` (using resource).
- **Scheduler state:** Either `fr` (free) or `a` (allocating).
- **Request flags:** Boolean variables indicating which processes have requested access.

Agents act as follows:

- Each process can: `id` (do nothing), `rq` (request), `rn` (renegue), or `rl` (release), depending on its current state.
- The Scheduler chooses one waiting process to grant access, using actions `p1` through `pn` if the corresponding process has requested access.

The property to be verified expresses fairness for each process: if a process is waiting, it will eventually no longer be waiting (i.e., it either gets access or gives up). This is encoded in SL[1G] as:

$$\#PR\langle\langle x\rangle\rangle[\![y_1]\!][\![y_2]\!]\cdots[\![y_n]\!](\text{Environment}, x)(P_1, y_1)\cdots(P_n, y_n)\bigwedge_{i=1}^{n}\mathsf{G}(wt_i \rightarrow \mathsf{F}\neg wt_i)$$

We translate this into an equivalent ATL* formula:

$$\bigwedge_{i=1}^{n} \langle\langle P_i \rangle\rangle \, \mathsf{G}(wt_i \rightarrow \mathsf{F}\neg wt_i)$$

Under the shared assumption of perfect information and recall, this translation is sound and preserves the intended fairness semantics for each process.

### 7.5.2 Model Encoding and Tool Differences

To ensure comparability, we use the benchmark ISPL models provided with the SL[1G] model checker. These are defined using local agent modules and are automatically flattened into global interpreted systems by MCMAS when requiring perfect information. In contrast, our ATL* model checker expects a fully flattened global-state representation as input since we always have the assumption of perfect information. We therefore generate equivalent global-state ISPL files for use in our tool, preserving agent behaviors and action availability.

We compare against the optimised SL[1G] mode in MCMAS, which splits the top-level SL[1G] formula into independent goals and generates a parity game for each [16]. This structure is closest to our recursive labelling approach in ATL*, where each strategic subformula is verified independently.

The results of this comparison are presented below.

### 7.5.3 Results

**Figure 7.8** presents the runtime comparison between the SL[1G] model checker and our parity-game–based ATL* model checker, as the number of processes in the scheduler benchmark increases from 1 to 6. A logarithmic scale is used to accommodate the wide variation in runtimes.

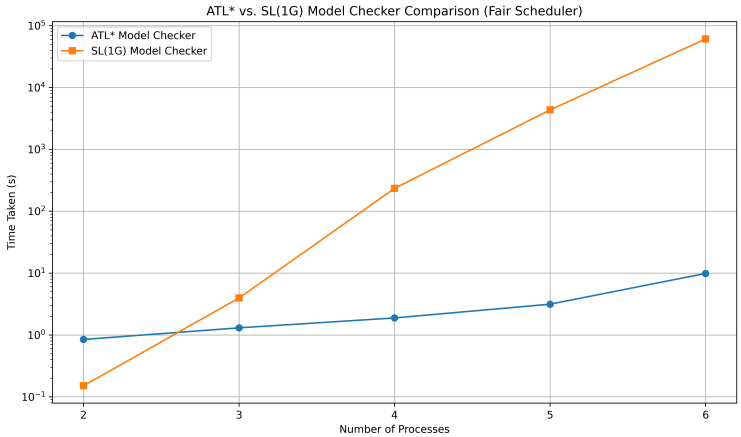


**Figure 7.8:** Runtime comparison between SL[1G] and ATL* model checkers as the number of processes increases. Logarithmic scale used.

The ATL* model checker outperforms SL[1G] for all cases with $n \geq 2$, with the difference becoming increasingly pronounced as the model size grows. At larger values of $n$, the performance gap spans

several orders of magnitude. For $n = 1$, the SL[1G] tool is marginally faster, likely due to fixed overheads in file I/O and symbolic setup that dominate runtime for very small parity games. Both tools exhibit exponential growth, indicated by the linear slope in the log-scale plot, but the ATL* checker shows a significantly flatter gradient, confirming superior scalability.

Several factors contribute to this improvement. First, our tool leverages modern external components for both DPA generation and parity game solving. In particular, the DPA is constructed using the Safra-less translation through Limit-Deterministic Büchi Automata (LDBA) [33], which has been shown to produce smaller automata in practice than the traditional approach implemented in the SL[1G] model checker based on nondeterministic Generalised Büchi automata. For game solving, we rely on parity solvers that implement quasi-polynomial time algorithms, which outperform the older symbolic implementation of Zielonka's algorithm used. Second, the logic itself plays a role: ATL* formulas are less expressive than full SL[1G] and result in structurally simpler and more compact parity games. These combined improvements in both algorithmic backend and logic design result in significantly better performance and scalability in practice.

These results clearly demonstrate that a dedicated ATL* model checker, tailored to the specific structure of the logic and leveraging modern automata tools, can significantly outperform general-purpose tools for more expressive logics, even when they share the same theoretical complexity class.

## 7.6 Evaluation Summary

This chapter presented a comparative evaluation of the two implemented ATL* model checking algorithms using the multi-agent counter benchmark. The parity-game–based approach significantly outperformed the Rabin tree automata–based approach, which was found to be impractical due to state space explosion during the non-emptiness check, even on small models.

We then compared the parity-game approach against the finite-trace model checking algorithms developed earlier in the thesis. These experiments confirmed the substantial performance and scalability benefits of finite-trace verification, driven by simpler automata constructions and more tractable model-checking procedures.

Finally, we compared our ATL* model checker to an existing SL[1G] model checker, demonstrating that a dedicated tool for ATL*, built around state-of-the-art automata and parity game solvers, can achieve orders-of-magnitude improvements over general-purpose tools for more expressive logics. These results highlight the effectiveness and practical viability of our algorithm for verifying multi-agent systems with ATL* properties.

# Chapter 8

# Conclusions and Future Work

## 8.1 Conclusions

The growing adoption of AI agents and multi-agent systems in security-critical, autonomous, and strategic environments makes formal verification more essential than ever. In particular, model checking of multi-agent logics such as ATL$^*$ provides rigorous guarantees about the strategic capabilities of agents in complex systems. This project contributes to this goal by developing, implementing, and evaluating scalable model checking algorithms for ATL$^*$ over both finite and infinite traces.

The project focused on two core verification problems: model checking ATL$^*_f$, the finite-trace variant suitable for terminating behaviours, and ATL$^*$, the classical infinite-trace semantics for reasoning over ongoing system executions. For ATL$^*_f$, we designed and implemented both an explicit-state and a symbolic-state model checker. The symbolic version is based on a novel adaptation of the automata-theoretic approach to a symbolic setting, introducing a fully symbolic product construction and fixpoint algorithm for ATL$^*_f$. The resulting model checker showed a substantial scalability advantage, verifying models with millions of states efficiently. This was demonstrated not only in synthetic benchmarks (Chapter 5) but also in a detailed, realistic cybersecurity case study (Chapter 6). These results confirm both the expressiveness of ATL$^*_f$ for goal-oriented multi-agent scenarios and the effectiveness of symbolic techniques for handling state space explosion.

In the infinite-trace setting, two algorithmic approaches were evaluated. The first was based on the classical automata-theoretic construction using Rabin tree automata, combining execution-tree automata for the CGS with deterministic Rabin automata for the formula. This approach required non-trivial reinterpretation of theoretical constructions to support symbolic data structures, but still suffered from combinatorial blowup due to state annotations, rendering it impractical for large-scale verification (see Section 7.2). The second approach reformulates ATL$^*$ model checking as a two-player parity game between the coalition and its complement. This parity-game-based algorithm exhibited much superior performance than the Rabin-tree-based approach, scaling to significantly larger models and handling a broad range of formulas efficiently, as demonstrated in Section 7.3.

Furthermore, our parity-game algorithm also outperformed the SL[1G] model checker in MCMAS on the fair scheduler synthesis benchmarks (Section 7.5). While SL[1G] has the same theoretical-complexity as ATL$^*$, the results demonstrate that using a logic-specific verification engine rather than relying on a more general-purpose tool can offer major practical advantages. By designing

algorithms that target ATL* directly, we avoid unnecessary overhead and can better exploit its structural properties and semantic requirements.

All of the algorithms developed in this work share the same theoretical worst-case complexity (2EXPTIME-complete), primarily due to the translation of temporal formulas to automata. To handle this efficiently, we integrated external tools for automata generation: in the finite case, a parallelised translation strategy leveraging multiple translators, and in the infinite case, recent Safra-less techniques. As shown in Sections 5.3 and 7.3, this modular translation strategy significantly improved performance and robustness across formulas of varying sizes.

An important insight from the evaluation is that theoretical complexity does not fully predict practical performance. Although all approaches are 2EXPTIME-complete, their implementability differs substantially depending on symbolic representations, automata structure, and game-solving techniques. The symbolic representation of the state space was crucial for the success of the finite-trace model checker, and the move from Rabin to parity-based reasoning in the infinite case resulted in major performance gains. These results are also partially justified by the concrete theoretical bounds derived in Sections 4.1.2 and 4.2.5.

In summary, the results clearly support the use of finite-trace verification when possible, due to its simpler automata and more tractable verification procedures. The resulting model checking tool integrates both of the ATL$^*_f$ algorithms and the parity-game-based ATL* checker, providing efficient and scalable verification across both semantic domains.

## 8.2 Future Steps

There are, however, steps that could be taken to improve the quality of the final product, as well as further avenues of research that could be explored to evaluate their effect on the performance of the checkers. These focus on improving the usability of all tools through the input format and potential algorithmic changes to improve the performance of both the Rabin tree automata and parity-game approach to model checking ATL*.

- **Extended ISPL Input Format**. While the current ISPL format is sufficient to successfully represent models, it could be extended to handle additional constructs. These include integer variables, conditional transitions and multiple state variables, all of which are available in the standard ISPL syntax. The parsing for these constructs could be re-used from MCMAS, although it would have to be adapted for the perfect information semantics and global states and evolution that we require in our models. This would not affect the performance of the model checkers, but it would lead to an improved user experience, requiring less effort from the user in modelling scenarios to be verified.

- **Improvements to the NRTA Non-Emptiness Algorithm**. The algorithm for NRTA non-emptiness was implemented explicitly due to the difficulties in performing the dualisation steps on the transition function symbolically. However, function-level performance measurements (see **Figure 7.3**) show that one of the two functions dominating execution time is outside the Rabin pair elimination loop, specifically the transformation from a WAA to a Simple WAA. This suggests that a conversion back into symbolic form at the point where we obtain a WAA with $k = 0$ Rabin pairs could improve the performance of this step. In fact, a symbolic implementation would potentially remove the need for this step, since an alternative to the

stack labelling algorithm in the last step could be used to remove the structural requirement of simple transition functions. Although this conversion would add overhead in terms of time and BDD variable usage, the potential performance benefits could justify the additional effort and lead to a more efficient approach to ATL* model checking through Rabin tree automata.

Alternatively, a more ambitious but potentially more scalable improvement would be to replace the explicit implementation of the NRTA non-emptiness algorithm with a formulation based on fixpoint equations. These equations, inspired by those presented in [50] for non-deterministic parity tree automata, avoid the need to explicitly construct intermediate weak alternating automata and perform annotations on states and transition functions. While the construction for the Rabin case is more complex and involves additional automata and steps thus making the adaptation to fixpoint equations non-trivial, developing such equations could yield an efficient, fully symbolic implementation of the algorithm and significantly enhance the performance of the ATL* model checking process.

- **Symbolic Parity Game Solving Algorithms**. Most of the research in solving parity games focuses on explicit algorithms, and hence most parity solvers including the one used in this project, Oink, are explicit. However recent papers ([20]) have presented set-based symbolic algorithms with quasi-polynomial performance, although this is measured in terms of the number of symbolic operations. This leads to a potential improvement to the parity-game-based approach to model checking ATL$^*$ by replacing the parity game solving through Oink with a custom implementation of one of these algorithms. This would enable a fully symbolic solution, without requiring the conversion into an explicit format to send as input to Oink, and thus might result in better performance.

# Chapter 9

# Declarations

## 9.1 Use of Generative AI

I acknowledge the use of GPT 4 (OpenAI, https://chat.openai.com/) for getting feedback on some parts of this report, and for fixing C++ compilation issues in the development of the code, as well as for summarising and understanding some of the papers read through the course of the project. I confirm that no content generated by AI has been presented as my own work.

## 9.2 Ethical Considerations

While this project does not raise direct ethical concerns, it contributes to the broader societal effort to develop trustworthy and verifiable AI. These goals are increasingly reflected in emerging regulatory frameworks, such as the UK's Bletchley Declaration on AI safety and transparency [39]. By advancing scalable verification techniques for agent-based systems, this work aligns with the ethical imperative to ensure that intelligent systems behave as intended, particularly in contexts where failures can lead to significant harm.

## 9.3 Sustainability

This project can be considered fairly sustainable since we did not use any GPUs for the benchmarking and evaluation steps, thus using less processing resources. The report was written completely online using the Latex editor Overleaf, which avoided the use of papers.

## 9.4 Data and materials

All the code for the model checking algorithms, the results from the evaluations and the modelling scripts used in Chapter 6 for the cyber-security scenario can be found in the following repo: `https://gitlab.doc.ic.ac.uk/sg2922/atlf_model_checker.git`

# Bibliography


[1] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. “Alternating-time temporal logic”. In: *Journal of the ACM* 49 (Sept. 2002), pp. 672–713. DOI: 10.1145/585265.585270. URL: https://dl.acm.org/doi/abs/10.1145/585265.585270.

[2] Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. *Jailbreaking Leading Safety-Aligned LLMs with Simple Adaptive Attacks*. 2024. URL: https://arxiv.org/abs/2404.02151.

[3] Tomáš Babiak et al. “Effective Translation of LTL to Deterministic Rabin Automata: Beyond the (F,G)-Fragment”. In: *Lecture Notes in Computer Science* (2013), pp. 24–39. DOI: 10.1007/978-3-319-02444-8_4. (Visited on 05/22/2025).

[4] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The Mit Press, 2008.

[5] Suguman Bansal, Yash Kankariya, and Yong Li. “DAG-Based Compositional Approaches for LTLf to DFA Conversions”. In: *Formal Methods in Computer-Aided Design* (2024). DOI: 10.34727/2024/isbn.978-3-85448-065-5. URL: https://repositum.tuwien.at/bitstream/20.500.12708/200795/1/Bansal-2024-DAG-Based%20Compositional%20Approaches%20for%20LTLf%20to%20DFA%20Conversions-vor.pdf.

[6] Suguman Bansal et al. “Hybrid Compositional Reasoning for Reactive Synthesis from Finite-Horizon Specifications”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34 (Apr. 2020), pp. 9766–9774. DOI: 10.1609/aaai.v34i06.6528. (Visited on 12/05/2024).

[7] Francesco Belardinelli et al. *Alternating-time Temporal Logic on Finite Traces*. July 2018. URL: https://www.ijcai.org/Proceedings/2018/0011.pdf.

[8] Raphaël Berthon, Bastien Maubert, and Aniello Murano. “Decidability Results for ATL* with Imperfect Information and Perfect Recall”. In: *Adaptive Agents and Multi-Agents Systems* (May 2017), pp. 1250–1258. DOI: 10.5555/3091125.3091299. (Visited on 05/20/2025).

[9] Dirk Beyer and Thomas Lemberger. “Software Verification: Testing vs. Model Checking”. In: *Hardware and Software: Verification and Testing* (2017), pp. 99–114. DOI: 10.1007/978-3-319-70389-3_7. (Visited on 05/05/2020).

[10] *Boost Graph Library: Adjacency List - 1.87.0*. Boost.org, 2025. URL: https://www.boost.org/doc/libs/1_87_0/libs/graph/doc/adjacency_list.html (visited on 03/24/2025).

[11] Charles L. Bouton. “Nim, A Game with a Complete Mathematical Theory”. In: *Annals of Mathematics* 3 (1901), pp. 35–39. DOI: 10.2307/1967631. URL: https://www.jstor.org/stable/1967631.

[12] Torben Braüner, Per Hasle, and Peter Øhstrøm. “Determinism and the Origins of Temporal Logic”. In: *Applied logic series* 16 (Jan. 2000), pp. 185–206. DOI: 10.1007/978-94-015-9586-5_10.

[13] Randal E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". In: *IEEE Transactions on Computers* C-35 (Aug. 1986), pp. 677–691. DOI: 10.1109/tc.1986.1676819.

[14] J.R. Burch et al. "Symbolic model checking: 1020 States and beyond". In: *Information and Computation* 98 (June 1992), pp. 142–170. DOI: 10.1016/0890-5401(92)90017-a. (Visited on 10/17/2019).

[15] Alberto Camacho et al. "Non-Deterministic Planning with Temporally Extended Goals: LTL over Finite and Infinite Traces". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31 (Feb. 2017). DOI: 10.1609/aaai.v31i1.11058. (Visited on 09/25/2022).

[16] Petr Cermák et al. "MCMAS-SLK: A Model Checker for the Verification of Strategy Logic Specifications". In: 8559 (2014), pp. 525–532.

[17] *Verifying and synthesising multi-agent systems against one-goal strategy logic specifications*. AAAI'15: Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence. Jan. 2015, pp. 2038–2044.

[18] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. "Alternation". In: *Journal of the ACM* 28 (Jan. 1981), pp. 114–133. DOI: 10.1145/322234.322243.

[19] Krishnendu Chatterjee, Thomas A. Henzinger, and Nir Piterman. "Strategy logic". In: *Information and Computation* 208 (June 2010), pp. 677–693. DOI: 10.1016/j.ic.2009.07.004. (Visited on 11/23/2020).

[20] Krishnendu Chatterjee et al. "Quasipolynomial Set-Based Symbolic Algorithms for Parity Games". In: *EPiC series in computing* 57 (Oct. 2018), pp. 233–211. DOI: 10.29007/5z5k. URL: https://arxiv.org/abs/1909.04983.

[21] Edmund Clarke et al. "Counterexample-Guided Abstraction Refinement". In: *Computer Aided Verification* 1855 (2000), pp. 154–169. DOI: 10.1007/10722167_15.

[22] Edmund M. Clarke and E. Allen Emerson. "Design and synthesis of synchronization skeletons using branching time temporal logic". In: *Logics of Programs* (May 1981), pp. 52–71. DOI: 10.1007/bfb0025774.

[23] Edmund M. Clarke, Orna Grumberg, and David E. Long. "Model checking and abstraction". In: *ACM Transactions on Programming Languages and Systems* 16 (Sept. 1994), pp. 1512–1542. DOI: 10.1145/186025.186051. (Visited on 01/09/2023).

[24] Costas Courcoubetis et al. "Memory-efficient algorithms for the verification of temporal properties". In: *Formal Methods in System Design* 1 (June 1990), pp. 275–288. DOI: 10.1007/bf00121128. (Visited on 04/24/2023).

[25] *CUDD: CU Decision Diagram Package Release 2.4.1*. Mit.edu, 2025. URL: https://web.mit.edu/sage/export/tmp/y/usr/share/doc/polybori/cudd/cuddIntro.html.

[26] Giuseppe De Giacomo and Moshe Y. Vardi. "Linear temporal logic and linear dynamic logic on finite traces". In: *IJCAI '13: Proceedings of the Twenty-Third international joint conference on Artificial Intelligence* (Sept. 2013), pp. 854–860. URL: https://dl.acm.org/doi/10.5555/2540128.2540252.

[27] Giuseppe De Giacomo and Moshe Y. Vardi. "Synthesis for LTL and LDL on finite traces". In: *IJCAI'15: Proceedings of the 24th International Conference on Artificial Intelligence* (July 2015), pp. 1558–1564. URL: https://dl.acm.org/doi/10.5555/2832415.2832466.

[28] Tom van Dijk. "Attracting Tangles to Solve Parity Games". In: *Lecture notes in computer science* 10805 (Jan. 2018), pp. 198–215. DOI: 10.1007/978-3-319-96142-2_14. (Visited on 05/23/2025).

[29] Tom van Dijk. "Oink: An Implementation and Evaluation of Modern Parity Game Solvers". In: *Lecture notes in computer science* 10805 (Jan. 2018), pp. 291–308. DOI: 10.1007/978-3-319-89960-2_16. (Visited on 05/23/2025).

[30] Ali Dorri, Salil S. Kanhere, and Raja Jurdak. "Multi-Agent Systems: A Survey". In: *IEEE Access* 6 (2018), pp. 28573–28593. DOI: 10.1109/access.2018.2831228.

[31] Alexandre Duret-Lutz et al. "Spot 2.0 — A Framework for LTL and $\omega$-Automata Manipulation". In: *Lecture Notes in Computer Science* 9938 (2016), pp. 122–129. DOI: 10.1007/978-3-319-46520-3_8. (Visited on 01/12/2025).

[32] E.Allen Emerson, Charanjit S. Jutla, and A.Prasad Sistla. "On model checking for the $\mu$-calculus and its fragments". In: *Theoretical Computer Science* 258 (Apr. 2001), pp. 491–522. DOI: 10.1016/S0304-3975(00)00034-7. URL: https://www.sciencedirect.com/science/article/pii/S0304397500000347.

[33] *From LTL and Limit-Deterministic Büchi Automata to Deterministic Parity Automata*. Vol. 10205. Tools, Algorithms for the Construction, and Analysis of Systems (TACAS 2017). Mar. 2017, pp. 426–442. URL: https://link.springer.com/chapter/10.1007/978-3-662-54577-5_25.

[34] Bernd Finkbeiner. *Automata, Games, and Verification: Lecture 15*. Feb. 2013. URL: https://finkbeiner.groups.cispa.de/teaching/automata-games-verification-12/downloads/notes15.pdf.

[35] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan Claypool Publishers, Jan. 2013. DOI: 10.1007/978-3-031-01564-9. (Visited on 01/12/2025).

[36] Giuseppe De Giacomo and Marco Favorito. "Compositional Approach to Translate LTLf/LDLf into Deterministic Finite Automata". In: *Proceedings of the International Conference on Automated Planning and Scheduling* 31 (May 2021), pp. 122–130. DOI: 10.1609/icaps.v31i1.15954. (Visited on 06/14/2023).

[37] Giuseppe De Giacomo et al. "Monitoring Business Metaconstraints Based on LTL and LDL for Finite Traces". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31 (Feb. 2014), pp. 1–17. DOI: 10.1007/978-3-319-10172-9_1.

[38] Valentin Goranko and Antje Rumberg. *Temporal Logic*. Ed. by Edward N. Zalta. Stanford Encyclopedia of Philosophy, 2020. URL: https://plato.stanford.edu/entries/logic-temporal/.

[39] UK Government. *The Bletchley Declaration by Countries Attending the AI Safety Summit, 1-2 November 2023*. GOV.UK, Nov. 2023. URL: https://www.gov.uk/government/publications/ai-safety-summit-2023-the-bletchley-declaration/the-bletchley-declaration-by-countries-attending-the-ai-safety-summit-1-2-november-2023.

[40] Aidan Harding, Mark Ryan, and Pierre-Ybes Schobbens. "Approximating ATL* in ATL". In: *VMCAI '02: Revised Papers from the Third International Workshop on Verification, Model Checking, and Abstract Interpretation* (Jan. 2002), pp. 289–301.

[41] David Harel and Amir Pnueli. "On the Development of Reactive Systems". In: *Logics and Models of Concurrent Systems* 13 (Feb. 1989), pp. 477–498. DOI: 10.1007/978-3-642-82453-1_17.

[42] Jesper G Henriksen et al. “Mona: Monadic second-order logic in practice”. In: *Lecture notes in computer science* (Jan. 1995), pp. 89–110. DOI: 10.1007/3-540-60630-0_5. (Visited on 11/30/2024).

[43] Gerard Holzmann, Eli Najm, and Ahmed Serhrouchni. “SPIN model checking: an introduction”. In: *International Journal on Software Tools for Technology Transfer (STTT)* 2 (Mar. 2000), pp. 321–327. DOI: 10.1007/s100090050039. (Visited on 04/29/2021).

[44] Gerard J Holzmann. *The SPIN model checker : Primer and reference manual*. Addison-Wesley Educational Publishers, 2011.

[45] John E Hopcroft. “An n log n algorithm for minimizing states in a finite automaton”. In: *Proceedings of an International Symposium on the Theory of Machines and Computations* (Sept. 1971), pp. 189–196. DOI: 10.1016/b978-0-12-417750-5.50022-1. (Visited on 01/16/2024).

[46] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley Longman, 2001.

[47] Michael Huth and Mark Ryan. *Logic in computer science : modelling and reasoning about systems*. Cambridge University Press, 2004.

[48] N.R. Jennings et al. “Automated Negotiation: Prospects, Methods and Challenges”. In: *Group Decision and Negotiation* 10 (2001), pp. 199–215. DOI: 10.1023/a:1008746126376.

[49] Jan Křetínský et al. “Rabinizer 4: From LTL to Your Favourite Deterministic Automaton”. In: *Lecture Notes in Computer Science* 10981 (2018), pp. 567–577. DOI: 10.1007/978-3-319-96145-3_30. (Visited on 05/22/2025).

[50] Orna Kupferman and Moshe Y Vardi. “Weak alternating automata and tree automata emptiness”. In: *STOC ’98: Proceedings of the thirtieth annual ACM symposium on Theory of computing* (May 1998). DOI: 10.1145/276698.276748.

[51] Orna Kupferman, Moshe Y. Vardi, and Pierre Wolper. “An automata-theoretic approach to branching-time model checking”. In: *Journal of the ACM* 47 (Mar. 2000), pp. 312–360. DOI: 10.1145/333979.333987. (Visited on 01/09/2022).

[52] John Lång and I. S. W. B. Prasetya. “Model checking a C++ software framework: a case study”. In: *ESEC/FSE 2019: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Sept. 2019), pp. 1026–1036. URL: https://doi.org/10.1145/3338906.3340453.

[53] Alessio Lomuscio, Hongyang Qu, and Franco Raimondi. “MCMAS: an open-source model checker for the verification of multi-agent systems”. In: *International Journal on Software Tools for Technology Transfer* 19 (Apr. 2015), pp. 9–30. DOI: 10.1007/s10009-015-0378-x. (Visited on 05/08/2023).

[54] Michael Luttenberger, Philipp J Meyer, and Salomon Sickert. “Practical synthesis of reactive systems from LTL specifications via parity games”. In: *Acta Informatica* 57 (Nov. 2019), pp. 3–36. DOI: 10.1007/s00236-019-00349-3. (Visited on 03/17/2024).

[55] Lockheed Martin. *Cyber Kill Chain*. Lockheed Martin, 2025. URL: https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html.

[56] Kenneth L McMillan. *Symbolic Model Checking*. Springer New York, NY, Jan. 1993. DOI: 10.1007/978-1-4615-3190-6. URL: https://doi.org/10.1007/978-1-4615-3190-6.

[57] Fabio Mogavero et al. “Reasoning About Strategies”. In: *ACM Transactions on Computational Logic* 15 (Aug. 2014), pp. 1–47. DOI: 10.1145/2631917. (Visited on 04/20/2022).

[58] *Alternating automata, the weak monadic theory of the tree, and its complexity*. 13th Proceedings of the International Colloquium on Automata, Languages, and Programming. 1986. URL: https://link.springer.com/chapter/10.1007/3-540-16761-7_77.

[59] Wonhong Nam and Hyunyoung Kil. "Formal Verification of Blockchain Smart Contracts via ATL Model Checking". In: *IEEE Access* 10 (2022), pp. 8151–8162. DOI: 10.1109/access.2022.3143145. (Visited on 02/05/2022).

[60] Doron Peled. "Ten years of partial order reduction". In: *Lecture Notes in Computer Science* 1427 (1998), pp. 17–28. DOI: 10.1007/bfb0028727. (Visited on 01/12/2025).

[61] Amir Pnueli. "The temporal logic of programs". In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)* (Sept. 1977). DOI: 10.1109/sfcs.1977.32. (Visited on 01/05/2022).

[62] Jean-Pierre Queille and Joseph Sifakis. "Specification and verification of concurrent systems in CESAR". In: *Proceedings of the 5th Colloquium on International Symposium on Programming* (Apr. 1982), pp. 337–351. URL: https://dl.acm.org/doi/10.5555/647325.721668.

[63] M. O. Rabin and D. Scott. "Finite Automata and Their Decision Problems". In: *IBM Journal of Research and Development* 3 (Apr. 1959), pp. 114–125. DOI: 10.1147/rd.32.0114. (Visited on 03/26/2020).

[64] Michael Rabin. "Decidability of Second-Order Theories and Automata on Infinite Trees". In: *Source: Transactions of the American Mathematical Society* 141 (1969), pp. 1–35. URL: https://lara.epfl.ch/w/_media/sav08/rabin69s2s.pdf (visited on 05/23/2025).

[65] Alur Rajeev et al. "MOCHA: Modularity in Model Checking". In: *Computer Aided Verification* 1427 (Jan. 1998).

[66] Scott Rose et al. "Zero trust architecture". In: *NIST Special Publication 800-207* (Aug. 2020). DOI: 10.6028/nist.sp.800-207. URL: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf.

[67] Shmuel Safra. "On the complexity of omega -automata". In: *Foundations of Computer Science* (Oct. 1988). DOI: 10.1109/sfcs.1988.21948. (Visited on 05/01/2023).

[68] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. "Smart Contract: Attacks and Protections". In: *IEEE Access* 8 (2020), pp. 24416–24427. DOI: 10.1109/ACCESS.2020.2970495. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8976179.

[69] Sven Schewe and Thomas Varghese. "Determinising Parity Automata". In: *Lecture Notes in Computer Science* 8634 (2014), pp. 486–498. DOI: 10.1007/978-3-662-44522-8_41. URL: https://arxiv.org/abs/1401.5394 (visited on 05/23/2025).

[70] Deian Tabakov and Moshe Y. Vardi. "Experimental Evaluation of Classical Automata Constructions". In: *Lecture Notes in Computer Science* (2005), pp. 396–411. DOI: 10.1007/11591191_28. URL: https://www.semanticscholar.org/paper/Experimental-Evaluation-of-Classical-Automata-Tabakov-Vardi/5d9dcd383f69b85018b809550eb8092eed899f0e.

[71] Ming-Hsien Tsai et al. "State of Büchi Complementation". In: *Lecture notes in computer science* 6482 (Jan. 2011), pp. 261–271. DOI: 10.1007/978-3-642-18098-9_28. (Visited on 01/12/2025).

[72] Moshe Y Vardi. "An automata-theoretic approach to linear temporal logic". In: *Lecture Notes in Computer Science* 1046 (Apr. 1996), pp. 238–266. DOI: 10.1007/3-540-60915-6_6. (Visited on 04/22/2023).

[73] Moshe Y Vardi. “Automata-Theoretic Model Checking Revisited”. In: *Verification, Model Checking and Abstract Interpretation* (Nov. 2007), pp. 137–150. DOI: 10.1007/978-3-540-69738-1_10. (Visited on 01/12/2025).

[74] Steen Vester. “On the Complexity of Model-Checking Branching and Alternating-Time Temporal Logics in One-Counter Systems”. In: *Lecture Notes in Computer Science* (2015), pp. 361–377. DOI: 10.1007/978-3-319-24953-7_27. URL: https://backend.orbit.dtu.dk/ws/files/118019604/main.pdf (visited on 05/23/2025).

[75] whitemech. *GitHub - whitemech/finite-synthesis-datasets: Datasets for Finite Synthesis.* GitHub, 2021. URL: https://github.com/whitemech/finite-synthesis-datasets/tree/main (visited on 01/12/2025).

[76] whitemech. *GitHub - whitemech/LTLf2DFA: From LTLf / PPLTL to Deterministic Finite-state Automata (DFA)*. GitHub, June 2021. URL: https://github.com/whitemech/LTLf2DFA.

[77] Jacob Wiebe, Ranwa Al Mallah, and Li Li. *Learning Cyber Defence Tactics from Scratch with Multi-Agent Reinforcement Learning*. arXiv.org, Sept. 2023. URL: https://arxiv.org/abs/2310.05939.

[78] Shufang Zhu et al. “Symbolic LTLf Synthesis”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence* (2017), pp. 1362–1369. URL: https://www.ijcai.org/proceedings/2017/0189.

[79] Wieslaw Zielonka. “Infinite games on finitely coloured graphs with applications to automata on infinite trees”. In: *Theoretical Computer Science* 200 (June 1998), pp. 135–183. DOI: 10.1016/s0304-3975(98)00009-7. (Visited on 03/15/2023).

# Appendix A

# Preliminary Results on LTL$_f$-to-DFA Conversion

As we have seen in Section **??**, the conversion of LTL$_f$ formulas into deterministic finite automata (DFA) is the most computationally expensive part of the ATL$^*_f$ model-checking algorithm. This step has a theoretical worst-case complexity of double exponential time with respect to the size of the input formula [27]. Nevertheless, it has been observed that the resulting DFA is often manageable in practice, particularly when determinisation is applied to automata over finite words ([70]).

Given the importance of this transformation step, extensive research has been done on developing tools that optimise the process and reduce its practical implementation overhead. In this section, we review the current state-of-the-art tools for LTL$_f$-to-DFA transformation and present the results of an experimental comparison to evaluate their efficiency and scalability.

## A.1 Overview of the Tools for LTL$_f$ to DFA Conversion

To approach the computational challenges of translating LTL$_f$ formulas into DFA, various tools have been developed, each employing different methodologies and intermediate structures. We provide an overview of five prominent tools – Spot [31], ltlf2dfa [76], Lisa[6], Lydia [36] and Lisa2 [5] – highlighting their underlying principles, methodologies and key optimisations.

Several LTL$_f$-to-DFA conversion tools use MONA, a specialised tool for transforming formulas in monadic second-order logic (MSO) – and hence first order logic (FOL) as well - into finite-state automata. MONA uses a semi-symbolic representation of the automata by leveraging shared multi-terminal Binary Decision Diagrams (shMTBDDs) to compactly represent transition functions (see section 2.2.1) [42].

**Spot**, introduced by Duret-Lutz et al., is one of the earliest tools developed for automata-based verification. It translates LTL$_f$ formulas into DFA by first converting them into equivalent LTL formulas, then to a nondeterministic Büchi automaton (NBA), which is determinised into a deterministic Büchi Automaton (DBA) and finally reduced to a DFA [31] However, the determinisation of Büchi automata and the complexity of handling temporal logics over infinite traces often make Spot slower and less scalable than newer approaches tailored specifically to finite traces.

**ltlf2dfa**, developed by Francesco Fuggitti, is a tool designed to translate LTL$_f$ formulas into deterministic finite automata (DFA). The translation process begins by converting the LTL$_f$ formula into an equivalent formula in First-Order Logic (FOL). The resulting FOL formula is then transformed into a DFA using the MONA tool [76]. The FOL-to-DFA transformation handled by MONA is theoretically nonelementary, meaning it can involve multiple nested exponentials in the complexity due to successive determinisations and projections introduced by handling quantifiers and negations [36]. However, this worst-case complexity rarely materialises, thus making ltlf2dfa a competitive tool for LTL$_f$-to-DFA conversion.

**Lisa** is a compositional tool that begins by unrolling the input LTL$_f$ formula to produce its abstract syntax tree (AST), splitting only on the outermost conjunctions. To manage the intermediate DFAs, Lisa employs a hybrid state-space representation, alternating between explicit-state representations and symbolic representations using reduced ordered BDDs. Explicit state is initially used as it is computationally efficient for small DFAs, but when minimisation becomes too expensive, Lisa transitions to symbolic representation to handle larger state spaces. This delays the use of symbolic operations as much as possible, therefore minimising the number of symbolic products required to generate the final DFA [6].

Another compositional method is **Lydia**, developed De Giacomo and Favorito. Lydia takes a fully compositional approach by unrolling the formula down to its propositional operators. It transforms propositional literals into DFA and uses automata operations specific to the operator being processed to compose the partial DFAs. Lydia uses Mona's shared multi-terminal BDD representation of intermediate DFAs to manage a larger state space. Although the worst-case complexity of this technique is again nonelementary, the aggressive minimisation performed on the partial automata ensures that this complexity rarely manifests in practice, making Lydia highly effective for real-world use cases [36].

**Lisa2**, the most recent LTL$_f$-to-DFA tool developed by Bansal et al., strikes a balance between the compositional approaches of Lisa and Lydia. Lisa2 unrolls formulas at the outermost Boolean operators, optimising the processing of sub-formulas while reducing composition steps. The tool introduces additional optimisations, such as duplication removal of sub-formulas in the AST and syntactic transformations to reduce the complexity of the input formula. Lisa2 supports both ROBDDs and shMTBDDs for representing the intermediate DFAs. This is due to each representation having advantages: shMTBDDs are faster but more memory intensive, whereas ROBDDs offer a slower but more memory-efficient alternative [5]. This flexibility enables Lisa2 to adapt to various computational constraints and problems.

## A.2 Experimental Results and Evaluation

To assess the performance of the tools described earlier and determine their suitability within the implementation of the ATL$^*_f$ model checking algorithm, a comprehensive evaluation was conducted, and the results are presented below.

### A.2.1 Experimental Setup

The evaluation was conducted on a Linux-based lab machine equipped with an Intel Core™ i7-10700 CPU at 2.90GHz, 16GB of RAM, and SSD storage. Each tool was executed with a time limit

of 3 minutes per test case, after which a timeout was reported. No explicit memory constraints were imposed during the experiments.

The following metrics were recorded for each test case: runtime – the time taken by each tool to complete the transformation and success/failure/timeout - whether the tool successfully converted the formula, failure to an error, or exceeded the time limit. The size of the DFAs returned was not recorded since all tools return minimal DFAs.

Benchmarks: the experiments used the Whitemech finite synthesis dataset [75], which includes a mix of structured and random benchmarks:

1. **SingleCounter**: A benchmark modelling a counter whose behaviour is entirely determined by the actions of the environment. The length of the formula corresponds to the maximum count, with n ranging from 1 to 20.

2. **Random**: conjunctions of n randomly generated LTLf formulas based on a set of basic cases, with 50 test cases for each n, where $3 \leq n \leq 10$ [78].

3. **Nim**: A generalised version of the game of Nim [11] with n heaps, where $1 \leq n \leq 20$, though only cases up to n = 5 were executed.

### A.2.2 Results

#### Single Counter Benchmark

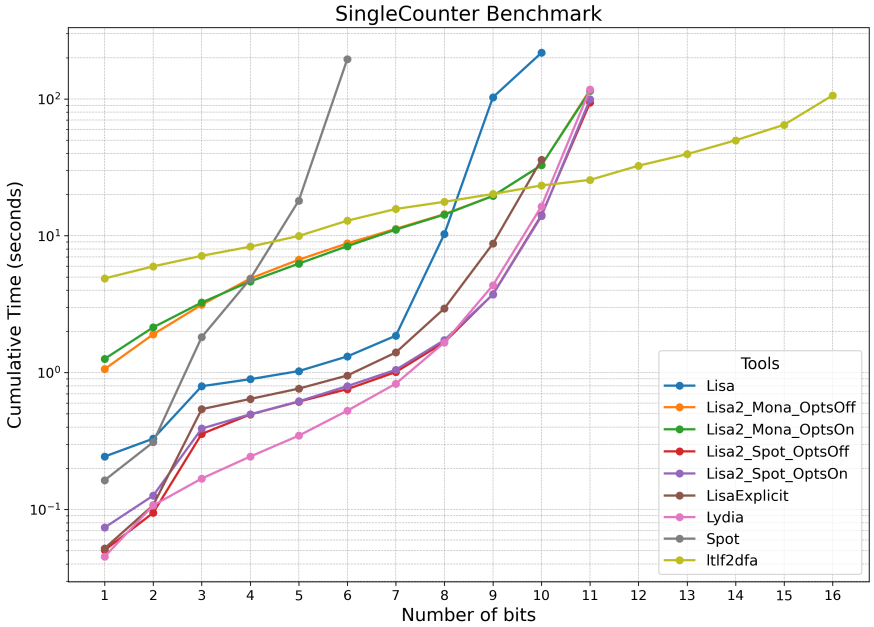


**Figure A.1:** Cumulative runtime for each formula length across all tools, excluding failed cases.

**FigureA.1** shows the cumulative average runtime for the SingleCounter benchmark. In this benchmark, ltlf2dfa achieves the best success rate, solving 80% (16/20) of cases, whereas the rest of the tools time out or fail with formulas larger than 11 bits, demonstrating its suitability for solving

complex structured benchmarks. However, in lower-bit formulas, ltlf2dfa is up to 10 times slower, resulting in a cumulative runtime worse than Lydia and Lisa2 for $n \leq 10$. For performance in the lower range, Lydia and Lisa2 (Spot representation) are the most efficient tools, consistently handling smaller formula sizes with lower runtimes.

### Random Benchmark

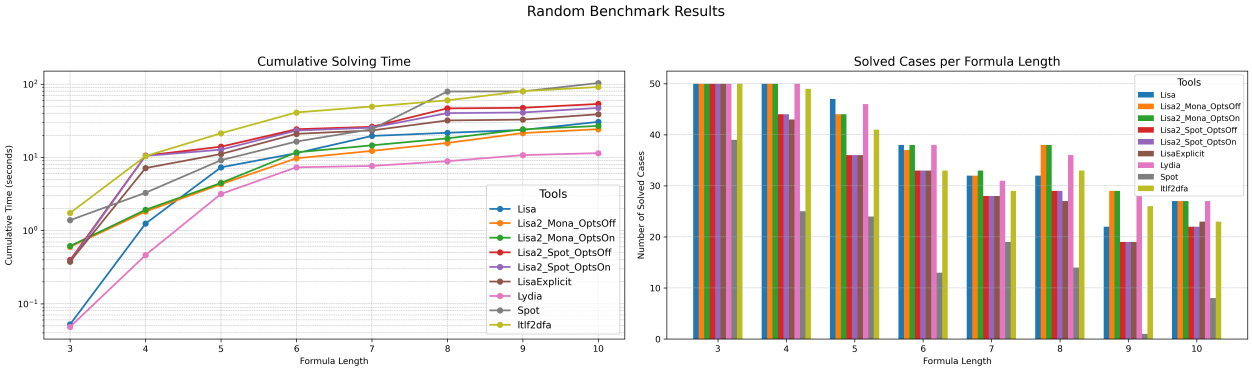


**Figure A.2:** Caption

**Figure A.2** (left) presents the cumulative average runtime for the random benchmark, while **Figure A.2** (right) shows the success rate for formulas of varying lengths. The tools that perform best in terms of runtime are Lisa2 (Mona representation) and Lydia, which handle increasing formula lengths efficiently. Again, ltlf2dfa is consistently about 10 times slower, reflecting its computational overhead, and in this case its resolution rate is not particularly better than that of other tools Lydia and Lisa2, which have similar performances in terms of resolution rate, although they show a significant drop in success rates, mostly due to timeout, declining to 50% by length 10.

The slower performance of Lisa2 (Spot representation) and LisaExplicit is notable, particularly in the lower length cases. This is likely due to the computational cost of determinising explicit state representations in LisaExplicit and the slower performance of ROBDDs in Lisa2 with Spot.

### Nim Benchmark

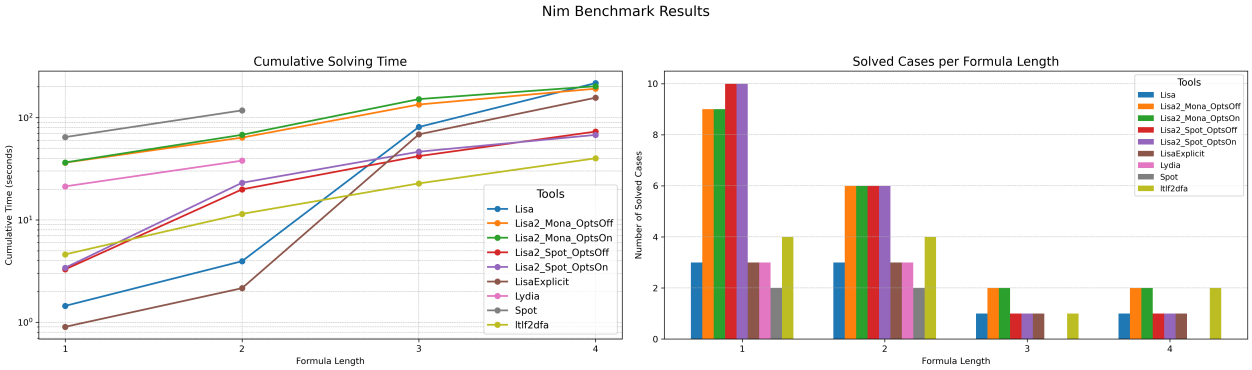


**Figure A.3:** Caption

The results for the Nim benchmark are depicted in **Figure A.3**. As shown, the success rate declines sharply for all tools as formula length increases, reflecting the inherent difficulty of encoding strategic game behavior. The best tools in terms of success rate are Lisa2 with Spot and Mona representations, which solve the most cases for the shorter formulas (lengths 1 and 2), though their performance drops significantly for longer formulas.

In terms of runtime, ltlf2dfa starts slower than most tools but demonstrates better scalability, maintaining a relatively stable growth compared to the steeper increases observed for Lisa2, Spot, and Lydia. By formula length 4, ltlf2dfa achieves the lowest cumulative runtime among tools that still solve a non-trivial number of instances, suggesting greater efficiency on complex inputs. Lydia, in contrast, performs poorly both in runtime and success rate, likely due to the cost of constructing and combining numerous intermediate DFAs in its full-decomposition strategy. While LisaExplicit and Lisa remain relatively efficient in runtime, their consistently low success rates limit their practical utility for this benchmark.

Focusing on the first two formula lengths where success rates are above 25% and thus more reliable, the best-performing tools in terms of runtime and coverage are the Lisa2 variants with Spot representation. However, the sharp decline in solved cases for these tools at higher lengths highlights the challenge of generalising their effectiveness to more complex game scenarios

**Overall Trends**

Several patterns emerge across the benchmarks:

1. **Spot**: the overhead of translating formulas through infinite-trace automata is evident in Spot's consistently poor performance in both runtime and success rate.

2. **Lisa** vs. **LisaExplicit**: Lisa with hybrid state space representations performs better and scales more effectively than its explicit state counterparts like LisaExplicit, although it perform worse than other more recent tools like Lydia or Lisa2.

3. **Lydia** and **Lisa2**: these tools exhibit the best runtime efficiency with Lisa2 (Spot representation) performing well in structured benchmarks whereas Lydia and Lisa2 (Mona representation) performing better in unstructured benchmarks (random dataset).

4. **Effect of Optimisations in Lisa2**: The choice of optimisation settings in Lisa2 (both for Spot and Mona representations) has only a marginal impact on both runtime performance and success rate, suggesting that its core performance characteristics are largely unaffected by these configuration options.

5. **Ltlf2dfa**: Despite its slower runtime, ltlf2dfa achieves the highest success rate of all tools, reliably solving difficult cases that other tools fail to handle within the 3-minute limit.

## A.3 Conclusion

The results demonstrate a clear trade-off between reliability, as observed with ltlf2dfa, and performance for smaller, more manageable formulas, where tools like Lydia and Lisa2 excel. Given the structure of formulas anticipated in model checking $ATL^*_f$, which aligns more closely with the characteristics of structured benchmarks, we decided to prioritise tools that perform well in these

benchmarks.

To balance both reliability and efficiency, we would like to explore an approach in which the implementation of the $ATL^*_f$ model checking algorithm uses a flexible parallel strategy. This would involve launching three tools in parallel: Lydia, for its exceptional performance in structured benchmarks like SingleCounter; Lisa2 with Spot, for its consistent results across structured benchmarks; and ltlf2dfa, to ensure reliable outcomes for complex formulas where the other tools may time out. The algorithm would then use the first result returned by any of the tools, enabling us to exploit the speed of Lydia and Lisa2 for simpler formulas, while maintaining the robustness of ltlf2dfa for more challenging cases. This approach aims to achieve an optimal balance between performance and reliability.

# Appendix B

# Rabin Tree Automata Approach Development

## B.1 Construction of the Execution-Tree Büchi Automaton

To decide whether $q \models \langle\langle A \rangle\rangle \psi$, we first build a nondeterministic Büchi tree automaton $\mathcal{A}_{S,q,A}$ whose language is exactly the set of $A$-controlled execution trees from $q$. The construction proceeds in three phases, as defined in [1]:

**1. Computing** $\mathrm{Post}(q, A)$**.** Let $\mathrm{Act}_A$ be the joint actions of coalition $A$. For each $\alpha \in \mathrm{Act}_A$, we symbolically compute

$$\rho_\alpha \;=\; \{\, q' \mid \exists\, \alpha' \in \mathsf{Act}_{-A} : \delta_{\mathcal{G}}(q, \alpha \wedge \alpha') = q' \,\}$$

as follows:

1. Conjoin the CGS transition BDD $\delta_{\mathcal{G}}(\vec{q}, \alpha, \vec{q}')$ with the encoding $\mathrm{enc}_c(q)$ of the current state.
2. Existentially abstract away $\vec{q}$ and the action-variables, yielding a BDD over $\vec{q}'$.
3. Reinterpret $\vec{q}'$ as the "current" variable set $\vec{q}$ via a `VectorCompose` operation.

We collect all such $\rho_\alpha$ into a candidate list, then minimise it by removing any $\rho$ that is a superset of another: symbolically, we drop $\rho_j$ if $\rho_j \implies \rho_i$ in BDD form. Finally, we extract each minimal $\rho$ into its constituent satisfying singleton-state BDDs assignments (one assignment per state) so that each $\rho$ becomes a $k$-tuple of next-state encodings.

**2. Automaton parameters.** We now have for every $q$ and $A$:

$$\mathrm{Post}(q, A) \;=\; \{\, \rho_1, \ldots, \rho_m \}, \quad \text{where } m \le |\mathrm{Act}_A|$$

Set

$$Q \;=\; S, \quad \Sigma \;=\; \Sigma_{\mathcal{G}}, \quad q_0 \;=\; q, \quad F \;=\; Q, \quad d \;=\; \max_i |\rho_i|.$$

**3. Transition function.** For each state $q$ and its labeling $\lambda(q)$, we define

$$\eta\big(q, \lambda(q)\big) \;=\; \bigvee_{\rho \in \mathrm{Post}(q,A)} \big(\mathrm{enc}_s(\rho)\big),$$

where if $\rho = \{s_1, \ldots, s_i\}$ (with $i \leq d$) then

$$\mathrm{enc}_s(\rho) \;=\; \big(\bigwedge_{j=1}^{i} \mathrm{enc}_{n,j-1}(s_j)\big) \;\wedge\; \big(\bigwedge_{j=i+1}^{d} \mathrm{enc}_{n,j-1}(s_i)\big),$$

duplicating the last state to fill all $d$ successor slots, which maintains correctness and ensures the branching factor is fixed. Here $\mathrm{enc}_{n,j}(s)$ binds the $j$th next-state variable set $\vec{q}^{\,j}$ to the singleton $\{s\}$. This yields the symbolic definition of $\mathcal{A}_{S,q,A} = \big\langle Q, \Sigma, \eta, q_0, F \big\rangle$.

## B.2 Formula DRTA Construction

The next step is to obtain a Deterministic Rabin Tree Automaton (DRTA) $\mathcal{A}_\psi$ that accepts exactly those infinite $d$-ary trees whose every branch satisfies the LTL formula $\psi$. We break this construction into three main phases:

**1. LTL → Deterministic Rabin Word Automaton** We first invoke **Rabinizer4** [49] to translate $\psi$ into a Deterministic Rabin word Automaton (DRA). Rabinizer4 employs a two-stage pipeline via deterministic generalised Büchi automata (DGBA), avoiding Safra's determinisation on NBAs [67] and yielding substantially smaller automata in practice [3, 71]. We process the external-tool output in HOA format, which specifies states, transitions $\Delta(q, a) = q'$, and transition-based Rabin acceptance pairs.

**2. Transition- to State-Based Acceptance** The remainder of our pipeline assumes state-based Rabin acceptance. We therefore apply Spot's `autfilt` tool [31] to convert the transition-based DRA into an equivalent state-based DRA:

$$\mathrm{Acc} \;=\; \{\, (G_1, B_1), \ldots, (G_k, B_k) \,\},$$

where each $G_i, B_i \subseteq Q$. In our implementation, we encode each $G_i$ and $B_i$ as a BDD over the current state variables of the DRTA, $\vec{d}$.

**3. Lifting to a $d$-ary Tree Automaton** Finally, we embed the deterministic word automaton into a fixed-degree $d$-ary tree automaton to align with the execution-tree automaton's branching structure. The state set, alphabet, initial state and acceptance condition remain the same. However, each word-automaton transition $\Delta(q, a) = q'$ is lifted into a transition

$$\delta(q, a) \;=\; \langle q', \ldots, q' \rangle \;\in Q^d,$$

i.e. we replicate the single successor $q'$ across all $d$ child-slots. Since the word automaton is deterministic, each $(q, a)$ has exactly one such $d$-tuple.

**Parsing and Integration** We implement a custom HOA parser similar to the one used in our finite-trace checker to read the DRA, extract $Q$, $\Sigma$, $\Delta$, and $\mathrm{Acc}$, and then perform the above lifting. The transition labels are encoded over the CGS proposition variables $\{p_j\}$ and the next and current states are encoded as in Section 4.1.1. Because the branching degree $d$ is determined by the execution-tree automaton, this lifting is performed once per state $q \in S$, whereas the translation is only done once for all the states.

## B.3 Product Automaton Construction

We form the synchronous product $\mathcal{A}_{\mathrm{prod}} = \mathcal{A}_{S,q,A} \otimes \mathcal{A}_{\psi}$, a $d$-ary tree automaton over the state space $S \times Q$ with initial state $(q, q_0^{\psi})$ and Rabin acceptance inherited from $\mathcal{A}_{\psi}$. Symbolically, its transition relation is obtained by conjoining the BDDs of the two component automata:

$$\delta_{\mathrm{prod}}(\vec{q}, \vec{d}, \vec{p}, \vec{q'}, \vec{d'}) \;=\; \delta(\vec{q}, \vec{p}, \vec{q'}) \;\wedge\; \Delta(\vec{d}, \vec{p}, \vec{d'}),$$

so that each satisfying assignment pairs the $i$th successor of the execution-tree automaton with the $i$th successor of the formula automaton.

To eliminate unreachable states, we lift the forward-reachability pruning from Section 3.3.2 to the product's $d$-ary successor relation: we compute the least fixpoint of reachable $(s, p) \in S \times Q$ from $(q_0, d_0)$, then restrict $\delta_{\mathrm{prod}}$ to transitions whose source and all $d$ successors lie in this reachable set. The resulting pruned BDD defines the final product automaton, ready for the emptiness check.

## B.4 NRTA Non-Emptiness Algorithm

Determining emptiness of the product NRTA $\mathcal{A}_{\mathrm{prod}}$ is the most intricate step of our pipeline. We follow Kupferman and Vardi's alternating-automata approach [50], which achieves the best-known bound for this problem of $O(n^{2k+1} \cdot k!)$ (where $n = |\mathcal{A}_{\mathrm{prod}}|$ and $k$ is the number of Rabin pairs). The core idea is to successively weaken the Rabin acceptance condition, encoding one pair at a time into an additional Büchi condition while alternating between dualisation steps, and finally reduce to emptiness of a weak alternating automaton.

Throughout this algorithm two key annotations are used on states of each intermediate alternating automaton constructed in Steps 3(a) and 3(c) in the algorithm below:

- **Pair index** $p \in \{1, \ldots, k\}$ indicates which Rabin pair $(G_p, B_p)$ the state in the automaton is currently enforcing.
- **Rank** $r \in [0, 2n]$ measures "distance" to acceptance along a run and guides the transition function transformations.

### B.4.1 Algorithm Overview

Below is a high-level overview of the five phases; full technical details can be found in [50, 51]. In the following section we focus on our own implementation structures and optimisations.

1. **Tree→ARA.** Flatten the $d$-ary NRTA into a one-letter alternating Rabin word automaton (ARA), at which point we convert from our symbolic BDD representation to an explicit positive-formula representation.

2. **ARA→WRAA.** Collapse all the ARA states into a single partition (Theorem 3.4 of [50]), yielding a weak Rabin alternating automaton (WRAA) with one global Rabin condition.
3. **Pair-elimination loop.**
   (a) Use Theorem 5.1 of [50] to encode the acceptance sets of all Rabin pairs $G_p$ into an additional Büchi $\gamma$, producing a weak Rabin Büchi alternating automaton (WRBAA) with $k-1$ pairs.
   (b) Dualise WRBAA (swap $\wedge/\vee$, TRUE/FALSE, and simple-state acceptance $\alpha$) to co-WRBAA.
   (c) Apply Theorem 5.2 of [50] to embed the now co-Büchi condition $\gamma$ into transitions, yielding a co-WRAA.
   (d) Dualise back to WRAA with one fewer Rabin pair.
4. **WAA conversion.** Once $k = 0$, reinterpret all remaining Rabin partitions as rejecting to obtain a pure weak alternating automaton (WAA).
5. **Emptiness of WAA.** Solve emptiness via the procedure of [51]:
   (a) Totalise the partition order of the WAA with Kahn's algorithm.
   (b) Rewrite transitions into "simple" binary formulas (Theorem 3.1) through subformula extraction
   (c) Perform the stack-based iterative labelling algorithm on the states to obtain the set of non-empty NRTA states (Theorem 4.7).

### B.4.2 Alternating Automaton Representation

We represent each intermediate alternating automaton $\mathcal{A} = (Q, \{a\}, \delta, \alpha, P)$ explicitly, with the transition function

$$\delta \colon Q \longrightarrow \mathcal{B}^+(Q)$$

encoded as shared directed acyclic graphs (DAGs) of positive Boolean formulas over state atoms. Concretely:

**State Structure** Each automaton state is stored in a lightweight struct containing the underlying BTA state ID and DRTA state ID, a small vector of pair indices $p \in [1..k]$ and a small vector of ranks $r \in [0..2n]$.

New annotations are simply pushed to the end of these vectors, so the most recent $p$ and $r$ are always at the back. It is necessary to keep the history of these annotations since we sometimes require previous annotations in the transition formula transformations.

**Formula DAGs** Transition formulas reside in a global hash-encoded DAG whose nodes are of five kinds:

$$\{\mathsf{TRUE},\ \mathsf{FALSE},\ \mathsf{ATOM}(q),\ \mathsf{AND},\ \mathsf{OR}\}.$$

Each AND or OR node holds a vector of child ids (FormulaIDs), and each ATOM node refers to a single state struct.

To ensure maximal sharing of subformulas, we use a global hash-encoding scheme: each time

we construct a new DAG node, we compute a key consisting of its node type (TRUE, FALSE, ATOM, AND, OR) together with its child FormulaIDs, and look it up in a hash map from these keys to existing FormulaIDs; if the key is present, we simply reuse the existing ID, otherwise we allocate a fresh node and record it in the map. This memoisation not only reduces memory usage by collapsing identical substructures, but also accelerates all subsequent transformations such as dualisation and annotation substitution since shared subformulas are processed only once. With this representation, key operations become simple recursive traversals:

- **Dualisation**: swap every AND $\leftrightarrow$ OR and TRUE $\leftrightarrow$ FALSE.
- **Annotation substitution**: replace $\mathsf{ATOM}(q)$ with a small formula encoding a disjunction or conjunction over several annotated versions of $q$.

### B.4.3 Symbolic-to-Explicit ARA Conversion

Emptiness of a nondeterministic tree automaton can be reduced to the one-letter emptiness problem for alternating automata [64, 51]. Accordingly, we convert our BDD-encoded NRTA product into an explicit one-letter alternating Rabin word automaton by the following steps.

First, we traverse each transition BDD and decode satisfying assignments for the current-state variables $\vec{q}$ and the $d$ successor-state vectors $\vec{q^0}, \ldots, \vec{q^{d-1}}$ into integer identifiers $(s, d)$ using binary operations. Next, for each source $(s, d)$, we assemble its transition formula in our shared DAG representation: each successor tuple $(s_0, \ldots, s_{d-1})$ becomes the conjunction $\bigwedge_{i=0}^{d-1} \mathsf{ATOM}(s_i)$, and we disjoin these over all tuples $\rho \in \delta_{\text{prod}}(s, d)$ to yield

$$\delta_{\text{ARA}}(s, d) \;=\; \bigvee_{\rho \in \delta_{\text{prod}}(s,d)} \Big( \bigwedge_{s' \in \rho} \mathsf{ATOM}(s') \Big).$$

We parse each Rabin acceptance pair $(G_i, B_i) \subseteq Q$ into two bitsets, enabling $O(1)$ membership tests; an additional Büchi acceptance is added as a single bitset, initially empty. Finally, since the alternating automaton operates over a singleton alphabet, we discard all original proposition labels and assume every transition is labeled by the dummy symbol $a$. The result is an explicit ARA whose transition map $Q \to \mathcal{B}^+(Q)$ uses FormulaIDs in our DAG, and whose acceptance vector of $k$ Rabin pairs (plus the Büchi set) fully encodes the original NRTA's acceptance conditions.

### B.4.4 Simple WAA Conversion

After eliminating all Rabin pairs in Step 4, we obtain a weak alternating automaton (WAA) whose transitions are arbitrary DAGs. To apply the stack-based emptiness algorithm, we require each transition to be in "simple" form: $\theta_1 \wedge \theta_2$ or $\theta_1 \vee \theta_2$, with each $\theta_i$ an ATOM. We therefore apply the following:

1. **Binary flattening.** Recursively rewrite every AND/OR node with more than two children into a binary tree of $\wedge/\vee$ nodes, introducing no new subformulas, simply changing the structure.
2. **Subformula extraction.** Perform a depth-first traversal of each transition DAG to collect all non-atomic subformulas. For each unique subformula $\phi$, create a fresh automaton state $q_\phi$ and redirect occurrences of $\phi$ in parent formulas to $\mathsf{ATOM}(q_\phi)$.
3. **Partition-aware ordering.** Assign each $q_\phi$ to the same weak partition index as its "parent" state with the lowest partition, ensuring the WAA's total order on partitions is respected.

Although this step can be expensive due to the high number of subformulas our shared-DAG representation, combined with hash-based caching of extracted subformulas, attempts to contain the blowup and make the final WAA emptiness more tractable in practice.

Once we have obtained a Simple WAA, we apply the stack-labelling algorithm in Theorem 4.7 of [51] to label each state in the simple WAA by `true` or `false`. If the initial state is labelled by `true`, we can conclude that the NRTA is non-empty and thus $G, q \models \langle\langle A \rangle\rangle \psi$ holds, i.e. the formula is true at state $q$. By applying this entire procedure to every state $q \in Q$ we obtain the set of states for which the formula holds.