Symbolic Verification for Concurrent Stochastic Games

Tang Yu Han, Brandon

A dissertation submitted for the degree of
Master of Computer Science (MCompSci)
Trinity 2026

Word Count: 10000

Abstract

Concurrent stochastic games (CSGs) are a formal model for multi-agent systems in which agents make decisions simultaneously and outcomes depend probabilistically on their joint actions. Probabilistic model checking of CSGs enables the verification of safety, liveness, and performance properties of such systems. Despite substantial existing work on CSG model checking, scalability remains a major challenge, with explicit-state techniques struggling to handle large state spaces.

Inspired by the success of symbolic verification for other probabilistic models, such as Markov decision processes and turn-based stochastic games, we develop novel symbolic algorithms for constructing symbolic representations of CSGs and verifying them against a range of two-coalition zero-sum properties. A key challenge in this setting is the computation of minimax values for collections of two-player zero-sum normal-form games. To address this, we present both a fully symbolic algorithm based on discounted regret matching and a hybrid approach using linear programming. To the best of our knowledge, this is the first symbolic model checking algorithm for CSGs.

We implement these techniques efficiently in the PRISM-games model checker and evaluate them on several case studies, including a profiling-based analysis of computational bottlenecks. Our results show substantial memory savings over existing explicit-state approaches in almost all cases, together with faster verification in many cases.

Declaration

In creating this work, I have used AI tools for the following tasks:

Coding: I used GitHub Copilot while writing Python scripts to test the correctness of my implementation and to run benchmarks.

Plotting and formatting: I used GitHub Copilot while writing Python scripts that formatted benchmark results into the tables in Chapter 8.

Spelling and grammar: I used ChatGPT to help address issues with grammar or clarity within parts of the thesis that seemed verbose or awkward.

Acknowledgements

I express my sincere gratitude to my supervisor, Professor David Parker, for his guidance, insight, and continued support throughout this project. I am also grateful to my friends for their encouragement and support during my studies, which made this journey far more manageable and enjoyable.

Contents

Chapter 1

Introduction

Probabilistic model checking [1, 2] is a formal verification technique for analysing systems that exhibit stochastic behaviour. Concurrent stochastic games [3, 4] (CSGs) provide a modelling framework for multi-agent systems in which agents make decisions simultaneously, and outcomes depend probabilistically on their joint actions. CSGs have been used to model a wide range of real-world systems, including communication protocols [5], security protocols [6, 7], and autonomous systems [8].

Verification of CSGs enables us to rigorously establish that a system satisfies specific quantitative properties. For example, one may verify that a process gains access to a shared resource within 10 seconds with probability at least 0.99, regardless of the actions taken by other processes. Probabilistic model checkers can both verify such properties and synthesise the strategies required to guarantee them.

Substantial theoretical work has been carried out on probabilistic model checking for CSGs, including both zero-sum [9, 10] and general-sum properties [8, 11]. Tool support has also been available for several years through PRISM-games [5]. However, scalability remains a major challenge. Existing implementations rely primarily on explicit-state techniques, representing state sets as bitsets and matrices/vectors as arrays, which limits the size of systems that can be analysed.

Symbolic model checking has achieved considerable success for non-probabilistic systems [12] and for simpler probabilistic models such as MDPs [13] and turn-based stochastic games (TSGs) [14]. In these contexts, decision-diagram representations are used to compactly encode state sets and transition relations. However, existing symbolic techniques do not directly extend to CSGs because model checking for CSGs requires solving a matrix game at each state, a process that is difficult to implement efficiently using standard decision-diagram operations.

To the best of our knowledge, this thesis presents and experimentally evaluates the first symbolic model checking algorithm for CSGs.

1.1. Challenges

The problem decomposes into two largely independent components: the symbolic construction of CSGs and the symbolic model checking of CSG properties.

While much of the existing theory and algorithms used in PRISM-games for other probabilistic models could, in principle, be reused, the modelling language for CSGs differs substantially from that of MDPs and TSGs. In particular, action labels define concurrent player actions rather than module synchronisation, and transitions may reference primed variables to enable coordinated updates across modules. These features necessitate the development of new algorithms for symbolic construction.

Standard CSG model checking algorithms rely on value iteration combined with the construction and solution of a matrix game at each state. In explicit-state implementations, these matrix games are typically solved using linear programming.

However, classical linear programming methods, such as the simplex algorithm, along with subroutines such as pivoting and matrix factorisation, are known not to work well with Algebraic Decision Diagrams (ADDs) as they involve data-dependent transformations that alter the structure of the matrices [15, 16].

This motivates the development of new symbolic minimax algorithms, or alternatively hybrid approaches that combine symbolic operations with explicit linear programming.

1.2. Contributions

This thesis develops novel algorithms for the construction and verification of symbolic models of CSGs, implements these techniques within the PRISM-games model checker, and evaluates them experimentally on a range of case studies.

In Chapter 4, we present a novel systematic procedure for transforming CSGs defined in the PRISM-games [5] language into a fully symbolic representation based on ADDs [15]. This includes the symbolic encoding of transition relations, action availability, and reward structures, while supporting features specific to PRISM-games CSG models such as idle actions and primed variables.

In Chapter 5, we develop new symbolic algorithms for model checking a range of two-coalition zero-sum probabilistic and reward-based properties expressed in rPATL. These algorithms are based on existing explicit-state techniques described in Section 3.3 but are

adapted to operate on symbolic representations, leveraging ADD operations for efficient computation.

These symbolic algorithms require, as a subroutine, the computation of minimax values for collections of two-player zero-sum normal-form games. To address this, we present two alternative implementations in Chapter 6 and Chapter 7. Chapter 6 gives a novel, fully symbolic formulation of discounted regret matching for computing minimax values for many two-player zero-sum normal-form games concurrently. This approach avoids explicit linear programming and does not require a fixed ADD variable ordering. Chapter 7 instead presents a hybrid algorithm that combines symbolic model construction with explicit linear programming for minimax computation, providing a practical alternative that performs better in most cases.

We develop an efficient implementation of the proposed techniques within the PRISM-games model checker using Java bindings to the University of Colorado Decision Diagram (CUDD) package [17]. This includes optimisations such as ADD minimisation via restriction [18] and dynamic variable reordering that have not been previously considered in PRISM-games.

We perform a detailed experimental evaluation on a variety of real-world case studies, comparing the time taken for different stages of the model checking process, and use profiling to identify bottlenecks in the algorithms. Our results demonstrate significant improvements in memory usage in almost all cases compared to existing explicit-state approaches, alongside highly competitive verification times that outperform explicit-state methods for checking probability-based properties.

Chapter 2

Related Work

This chapter situates our work relative to explicit-state model checking for CSGs and symbolic probabilistic verification for other stochastic models.

2.1. Explicit Model Checking for CSGs

Probabilistic model checking has been extensively studied for MDPs and stochastic games, with a mature body of algorithms for reachability and reward objectives [19]. For multi-agent systems, tool support has historically focused on TSGs due to their simpler optimisation structure [20], but CSGs have become increasingly important for modelling systems with simultaneous decisions [5].

For two-coalition zero-sum objectives, the standard approach is value iteration, where each iteration requires computing the minimax value of a two-player normal-form game induced at each state [10, 21, 22]. In practice, explicit implementations typically solve these matrix games via linear programming [1]. For infinite-horizon objectives, qualitative pre-computation based on (nested) fixed-point computations can accelerate convergence by identifying states with value 0 or 1 a priori [23, 24].

An alternative approach to value iteration is policy iteration (also known as strategy improvement). In this approach, one alternates between computing the values that result from a fixed strategy profile and improving the strategy profile based on the computed values [9, 25, 26]. However, this approach has less mature tool support in this setting and thus is not the focus of this thesis.

PRISM-games provides a widely used implementation of these techniques, supporting CSG modelling and verification for both zero-sum and more general solution concepts [5]. Despite these advances, explicit-state representations (bitsets for state sets; arrays/ matrices for vectors and transitions) remain a limiting factor for scalability on large models.

2.2. Symbolic Verification for DTMCs, MDPs and TSGs

Symbolic model checking represents large state spaces and transition relations compactly using decision diagrams [12]. For probabilistic systems, symbolic approaches based on ADDs have been particularly successful for discrete-time Markov Chains (DTMCs) and MDPs, combining symbolic model construction with verification algorithms expressed as decision-diagram operations (e.g., abstraction and matrix-vector multiplication) [13, 27].

This framework has also been extended to TSGs by incorporating state ownership into the symbolic transition representation and applying min/max operations according to the controlling player [28]. These results demonstrate that symbolic representations can substantially reduce memory consumption and enable verification of models beyond the reach of explicit-state engines.

2.3. Issues with Extending Symbolic Verification to CSGs

The key obstacle in extending symbolic verification from TSGs to CSGs is the optimisation structure of value iteration. In TSGs, the iteration reduces to a pointwise minimum or maximum over the enabled actions of a single player for each state, which maps naturally to symbolic abstraction. In CSGs, each iteration requires computing the minimax value of a normal-form game induced by the joint action choices of two coalitions [1, 21].

The standard method for computing minimax values uses linear programming; however, classical LP solvers (e.g., simplex) rely on pivoting and other data-dependent control flow that alters matrix structure in ways that do not align well with ADD-based symbolic computation [15, 16]. As a result, existing symbolic techniques for probabilistic verification do not immediately yield scalable symbolic CSG model checking.

2.4. Summary

Overall, explicit CSG model checking is well established but limited by explicit-state representations, while symbolic techniques scale successfully for DTMCs, MDPs and TSGs but do not directly address the matrix-game bottleneck of CSGs. This thesis addresses this gap by developing (i) symbolic construction procedures for PRISM-games CSG models and (ii) symbolic and hybrid methods for computing minimax values within value iteration, enabling symbolic verification for two-coalition zero-sum CSG properties.

Chapter 3

Preliminaries

3.1. Concurrent Stochastic Games

A concurrent stochastic game is an 8-tuple [7]

$$G = (N, S, S_{\text{initial}}, A, \Delta, \delta, \text{AP}, L)$$

where:

- $N = \{1, ..., n\}$ is a finite set of players
- S is a finite set of possible states of the game
- $S_{\text{initial}} \subseteq S$ is the non-empty set of initial states
- $A = A_1 \times ... \times A_n$ where
  - A_p is a finite set of actions of player $p \in N$ with $A_i \cap A_j = \emptyset$ for all $i \neq j$
  - and $\perp_p$ is the idle action for player p, with $\perp_p \in A_p$ for every $p \in N$
- $\Delta \subseteq S \times \cup_{i=1}^{n} A_i$ is a set defining available actions, where a_i is enabled in state s iff $(s, a_i) \in \Delta$
- $\delta : S \times A \to \text{Dist}(S)$ is a probabilistic transition function, i.e. $\delta(s, \vec{a})(s') \in [0, 1]$ represents the probability of transitioning to s' from s when $\vec{a}$ is taken
- AP is a set of atomic propositions
- $L : S \to 2^{\text{AP}}$ is a labelling function for defining properties

G starts in some $s \in S_{\text{initial}}$. At each state, each player $p \in N$ chooses some action $a_p \in A_p$ where $(s, a_p) \in \Delta$. If player p has no available non-idle actions ($a \neq \perp_p$) in state s, we say that p is stuck and must perform the idle action $\perp_p$. The idle action is available if and only if p is stuck. This guarantees that each player has at least one enabled action in every state.

The selection of actions $\vec{a} = (a_1, ..., a_n) \in A$ is the "action tuple" chosen, which results in a probabilistic transition to another state s' according to the distribution $\delta(s, \vec{a})$.

A strategy of player p is a function $\sigma_p : \text{FPaths}_G \to \text{Dist}(A_p)$ assigning a probability distribution over p's enabled actions based on the observed finite history of states and other players' actions. A strategy profile is a tuple of strategies, one for each player, i.e. $\sigma = (\sigma_1, ..., \sigma_n)$.

The set of possible strategies for player p is denoted as Σ_p, and the set of possible strategy profiles is $\Sigma = \Sigma_1 \times ... \times \Sigma_n$.

3.1.1. Reward Structures

We focus on CSGs that are extended with reward structures. A reward structure is a pair $(r^r_{\text{state}}, r^r_{\text{trans}})$ where:

$$r^r_{\text{state}} : S \to \mathbb{R}$$
$$r^r_{\text{trans}} : S \times A \to \mathbb{R}$$

The state reward function $r^r_{\text{state}}(s)$ gives the reward obtained when the game is in state s, while the transition reward function $r^r_{\text{trans}}(s, \vec{a})$ gives the reward obtained when the players select the action tuple $\vec{a}$ at state s. The superscript r is used to distinguish different reward structures.

We can define multiple reward structures in a CSG but they are used independently in properties and model checking.

3.2. Zero-Sum Properties of Concurrent Stochastic Games

Properties of CSGs are defined in the reward-extended probabilistic alternating-time temporal logic (rPATL) [7]. We will focus on a subset of rPATL that only contains two-coalition zero-sum properties.

3.2.1. Syntax

The syntax of rPATL is given by the following grammar:

$$\varphi := \text{true} \mid a \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle C \rangle\rangle P_{\sim q}[\psi] \mid \langle\langle C \rangle\rangle R^r_{\sim x}[\rho]$$
$$\psi := \mathtt{X}\varphi \mid \varphi\ \mathtt{U}^{\leq k}\ \varphi \mid \varphi\ \mathtt{U}\ \varphi$$
$$\rho := \mathtt{I}^{=k} \mid \mathtt{C}^{\leq k} \mid \mathtt{F}\varphi$$

where:

- a is an atomic proposition
- $C \subseteq N$ is a coalition of players
- $\sim \in \{<, \leq, \geq, >\}$
- $k \in \mathbb{N}, q \in \mathbb{R} \cap [0, 1], x \in \mathbb{R}$

For notational convenience, we define the *finally* and the *globally*[1] operators for path formulae:

$$\langle\langle C\rangle\rangle P_{\sim q}[\mathtt{F}\varphi] \coloneqq \langle\langle C\rangle\rangle P_{\sim q}[\text{true } \mathtt{U}\ \varphi]$$
$$\langle\langle C\rangle\rangle P_{\sim q}[\mathtt{G}\varphi] \coloneqq \langle\langle C\rangle\rangle P_{\sim q}[\neg(\mathtt{F}(\neg\varphi))]$$

3.2.2. Semantics

3.2.2.1. State Formulae

The syntax of rPATL distinguishes between state formulae (φ), path formulae (ψ) and reward formulae (ρ). State formulae are evaluated over states S of the game, while path and reward formulae are evaluated over infinite traces of the game. An infinite trace τ is an alternating sequence of states and action tuples, i.e. $\tau \coloneqq s_0, \vec{a}_1, s_1, \vec{a}_2, ...$ This corresponds to the game starting in state s_0, then action tuple $\vec{a}_1$ being taken, resulting in a transition to state s_1, and so on.

The satisfaction relation $\vDash$ for state formulae φ is defined inductively over the structure of φ. The propositional logic fragments $(\text{true}, a, \neg, \wedge)$ are defined in their usual way.

A state satisfies a formula $\langle\langle C\rangle\rangle P_{\sim q}[\psi]$ if the coalition of players C can ensure that the probability of the path formula ψ being satisfied is $\sim q$, regardless of the strategies of the other players $(N \setminus C)$.

A state satisfies a formula $\langle\langle C\rangle\rangle R^r_{\sim x}[\rho]$ if the players in C can ensure that the expected value of the reward formula ρ for reward structure r is $\sim x$, regardless of the strategies of the other players $(N \setminus C)$.

3.2.2.2. Path and Reward Formulae

We formalise the satisfaction of path and reward formulae via the coalition game G^C with respect to a fixed coalition C.

$$G^C = \left(\{C, N \setminus C\}, S, S_{\text{initial}}, \left(A_C, A_{N\setminus C}\right), \Delta^C, \delta, \text{AP}, L\right)$$

where:

- $A_D = \prod_{p\in D} A_p$ for $D \in \{C, N \setminus C\}$
- $(s, a_d) \in \Delta^C$ iff each a_i in a_d is a valid action by player i at state s

We then have

[1] Safety property proofs reduce to reachability property proofs, so we only focus on the latter.

$$\langle\langle C\rangle\rangle P_{\max=?}[\mathtt{G}\varphi](s) = \langle\langle C\rangle\rangle P_{\max=?}[\neg(\mathtt{F}(\neg\varphi))](s) = 1 - \langle\langle C\rangle\rangle P_{\min=?}[\mathtt{F}(\neg\varphi)](s)$$

$$s \vDash \langle\!\langle C \rangle\!\rangle P_{\sim q}[\psi] \Leftrightarrow \exists \sigma_C \in \Sigma_C \forall \sigma_{N \setminus C} \in \Sigma_{N \setminus C}. \mathbb{E}^{\sigma_C, \sigma_{N \setminus C}}_{G^C, s}(\psi(\tau)) \sim q$$

$$s \vDash \langle\!\langle C \rangle\!\rangle R^r_{\sim x}[\rho] \Leftrightarrow \exists \sigma_C \in \Sigma_C \forall \sigma_{N \setminus C} \in \Sigma_{N \setminus C}. \mathbb{E}^{\sigma_C, \sigma_{N \setminus C}}_{G^C, s}(\rho(\tau)) \sim x$$

where

$$\psi(\tau) = 1 \text{ if } \tau \vDash \psi \text{ and } 0 \text{ otherwise}$$

$$\rho(\tau) = \mathrm{rew}(r, \tau)(\rho)$$

The satisfaction of path formulae $\tau \vDash \psi$ and the random variable $\mathrm{rew}(r, \tau)(\rho)$ for a random trace $\tau = s_0, \vec{a}_1, s_1, \vec{a}_2, ...$ are defined inductively on the structure of ψ and ρ respectively.

A path formula ψ may take one of three forms: $\mathtt{X}\varphi$, $\varphi_1\ \mathtt{U}^{\leq k}\ \varphi_2$, or $\varphi_1\ \mathtt{U}\ \varphi_2$. These use the PCTL semantics [29]:

$$\tau \vDash \mathtt{X}\varphi \Leftrightarrow s_1 \vDash \varphi$$

$$\tau \vDash (\varphi_1\ \mathtt{U}^{\leq k}\ \varphi_2) \Leftrightarrow \exists_{0 \leq i \leq k} \big(s_i \vDash \varphi_2 \wedge \big(\forall_{0 \leq j < i}\ s_j \vDash \varphi_1\big)\big)$$

$$\tau \vDash (\varphi_1\ \mathtt{U}\ \varphi_2) \Leftrightarrow \exists_{i \geq 0} \big(s_i \vDash \varphi_2 \wedge \big(\forall_{0 \leq j < i}\ s_j \vDash \varphi_1\big)\big)$$

A reward formula ρ can include the instantaneous reward operator $\mathtt{I}^{=k}$, the cumulative reward operator $\mathtt{C}^{\leq k}$ and the final reward operator $\mathtt{F}\varphi$. These are defined as follows:

$$\mathrm{rew}(r, \tau)\big(\mathtt{I}^{=k}\big) := r^r_{\mathrm{state}}(s_k)$$

$$\mathrm{rew}(r, \tau)\big(\mathtt{C}^{\leq k}\big) := r(\tau(..k))$$

$$\mathrm{rew}(r, \tau)(\mathtt{F}\varphi) := \begin{cases} r(\tau(..k)) \text{ where } s_k \vDash \varphi \text{ and } \forall i < k\ (s_i \nvDash \varphi) & \text{if } \exists k \text{ s.t. } s_k \vDash \varphi \\ \infty & \text{otherwise} \end{cases}$$

where:

$$r(\tau(..k)) = \sum_{i=0}^{k-1} \big(r^r_{\mathrm{state}}(s_i) + r^r_{\mathrm{trans}}(s_i, \vec{a}_{i+1})\big)$$

We note that $\mathrm{rew}(r, \tau)\big(\mathtt{C}^{\leq k}\big)$ and $\mathrm{rew}(r, \tau)(\mathtt{F}\varphi)$ intentionally do not include $r^r_{\mathrm{state}}(s_k)$.

3.2.3. Quantitative Queries

In addition to the above properties that have a Boolean satisfaction value, we also have quantitative queries that ask for the maximum/minimum probability or expected reward a coalition can achieve.

As shown later in Section 3.3, this can be computed as part of the model checking procedure for the above properties.

$$\langle\langle C\rangle\rangle P_{\max=?}[\psi](s) = \max_{\sigma_C \in \Sigma_C} \min_{\sigma_{N\setminus C} \in \Sigma_{N\setminus C}} \mathbb{E}_{G^C,s}^{\sigma_C,\sigma_{N\setminus C}}(\psi(\tau))$$

$$\langle\langle C\rangle\rangle R^r_{\max=?}[\rho](s) = \max_{\sigma_C \in \Sigma_C} \min_{\sigma_{N\setminus C} \in \Sigma_{N\setminus C}} \mathbb{E}_{G^C,s}^{\sigma_C,\sigma_{N\setminus C}}(\rho(\tau))$$

Since we are dealing with zero-sum properties, we have the following duality:

$$\langle\langle C\rangle\rangle P_{\max=?}[\psi](s) = \langle\langle N \setminus C\rangle\rangle P_{\min=?}[\psi](s)$$

$$\langle\langle C\rangle\rangle R^r_{\max=?}[\rho](s) = \langle\langle N \setminus C\rangle\rangle R^r_{\min=?}[\rho](s)$$

3.3. Model Checking CSG Properties

Here, we give detailed presentations of algorithms for model checking the zero-sum properties described above, allowing comparison with their symbolic versions in Chapter 5. Given a property φ, we are interested in finding the function $V_G(\varphi) : S \to \mathbb{B}$. This function maps each state $s \in S$ to 1 if $s \vDash \varphi$ and 0 otherwise. We drop the subscript G when the game is clear from context.

These algorithms rely on solving matrix games as a subroutine (see Section 3.6). We define $\text{val}(Z)$ to be the minimax value of the matrix game defined by the payoff matrix $Z \in \mathbb{R}^{n\times m}$. In the existing explicit-state implementation, $\text{val}(Z)$ is calculated using linear programming.

As is typical in formal methods, we check properties recursively over the structure of the property's formula. The base cases true, a are straightforward, and the propositional logic operators $\neg, \wedge$ are handled in the usual way.

Deriving and proving these algorithms is beyond the scope of this thesis; instead, we focus on their execution details. More information about the theory behind these algorithms can be found in [30].

We consider the model checking of probabilistic properties, followed by reward properties.

3.3.1. Probabilistic Properties

To compute $V\big(\langle\langle C\rangle\rangle P_{\sim q}[\psi]\big)$, we first find $V(\langle\langle C\rangle\rangle P_{\max=?}[\psi]) : S \to [0,1]$ or $V(\langle\langle C\rangle\rangle P_{\min=?}[\psi]) : S \to [0,1]$ for all states, then compare the resulting probabilities with q using $\sim$.

We focus on computing $V(\langle\langle C\rangle\rangle P_{\max=?}[\psi])$, with $V(\langle\langle C\rangle\rangle P_{\min=?}[\psi])$ reducing to $V(\langle\langle N \setminus C\rangle\rangle P_{\max=?}[\psi])$ due to the duality mentioned above.

For path formulae ψ, we consider each case separately.

Next

This reduces to solving a single matrix game for each state:

$$V(\langle\langle C\rangle\rangle P_{\max=?}[\mathtt{X}\varphi])(s) = \mathrm{val}(Z(s))$$

with $Z(s)_{i,j} = \sum_{s' \in \mathrm{Sat}(\varphi)} \delta^C\big(s, \big(a_i, b_j\big)\big)(s')$ where a_i and b_j range over all possible actions for coalition C and $N \setminus C$ in state s respectively.

Bounded Until

We use value iteration to recursively compute $V(\langle\langle C\rangle\rangle P_{\max=?}[\varphi_1 \ \mathtt{U}^{\leq n} \ \varphi_2])$ for $0 \leq n \leq k$

$$V(\langle\langle C\rangle\rangle P_{\max=?}[\varphi_1 \ \mathtt{U}^{\leq n} \ \varphi_2])(s) = \begin{cases} 1 & \text{if } s \in \mathrm{Sat}(\varphi_2) \\ 0 & \text{else if } s \notin \mathrm{Sat}(\varphi_1) \\ 0 & \text{else if } n = 0 \\ \mathrm{val}(Z(s)^n) & \text{otherwise} \end{cases}$$

$$Z(s)^n_{i,j} = \sum_{s' \in S} \delta^C\big(s, \big(a_i, b_j\big)\big)(s') \times V(\langle\langle C\rangle\rangle P_{\max=?}[\varphi_1 \ \mathtt{U}^{\leq n-1} \ \varphi_2])(s')$$

where a_i and b_j range over all possible actions for coalition C and $N \setminus C$ in state s respectively.

Unbounded Until

Unbounded until properties are computed via value iteration to convergence. We use the same recurrence as for bounded until, but stop when the values converge within some small epsilon $\varepsilon > 0$.

Furthermore, we can also use qualitative reachability and safety pre-computations [24] to speed up convergence. These can be used to find the states S_1 and S_0 such that $V(\langle\langle C\rangle\rangle P_{\max=?}[\varphi_1 \ \mathtt{U} \ \varphi_2])(s) = 1$ or 0 directly, allowing us to only perform value iteration on the remaining states.

These are incorporated into the value iteration as follows:

$$V(\langle\!\langle C \rangle\!\rangle P_{\max=?}[\varphi_1 \ \mathtt{U} \ \varphi_2])(s) = \lim_{n\to\infty} v_n^s$$

$$v_n^s = \begin{cases} 1 & \text{if } s \in S_1 \\ 0 & \text{else if } s \in S_0 \\ 0 & \text{else if } n = 0 \\ \mathrm{val}(Z(s)^n) & \text{otherwise} \end{cases}$$

$$Z(s)_{i,j}^n = \sum_{s' \in S} \delta^C\big(s, (a_i, b_j)\big)(s') \times v_{n-1}^{s'}$$

3.3.2. Reward Properties

Similarly to probabilistic properties, to find states that satisfy $\langle\!\langle C \rangle\!\rangle R^r_{\sim x}[\rho]$, we first find $\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\rho]$ or $\langle\!\langle C \rangle\!\rangle R^r_{\min=?}[\rho]$ for all states, then compare the resulting expected rewards with x using $\sim$. We explain how to compute $\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\rho]$.

For reward formulae ρ, we consider each case separately.

Instantaneous Reward

We use value iteration to recursively compute the values of $V(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\mathtt{I}^{=n}])$ for $0 \leq n \leq k$

$$V(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\mathtt{I}^{=n}])(s) = \begin{cases} r^r_{\text{state}}(s) & \text{if } n = 0 \\ \mathrm{val}(Z(s)^n) & \text{otherwise} \end{cases}$$

$$Z(s)_{i,j}^n = \sum_{s' \in S} \delta^C\big(s, (a_i, b_j)\big)(s') \times V\big(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}\big[\mathtt{I}^{=(n-1)}\big]\big)(s')$$

Bounded Cumulative Rewards

We use value iteration to recursively compute the values of $V\big(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\mathtt{C}^{\leq n}]\big)$ for $0 \leq n \leq k$

$$V\big(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\mathtt{C}^{\leq n}]\big)(s) = \begin{cases} 0 & \text{if } n = 0 \\ r^r_{\text{state}}(s) + \mathrm{val}(Z(s)^n) & \text{otherwise} \end{cases}$$

$$Z(s)_{i,j}^n = r^r_{\text{trans}}\big(s, (a_i, b_j)\big) + \sum_{s' \in S} \big(\delta^C\big(s, (a_i, b_j)\big)(s') \times V\big(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}\big[\mathtt{C}^{\leq (n-1)}\big]\big)(s')\big)$$

Expected Reachability

While we allow negative rewards, we need the following assumption to ensure the correctness and convergence of the algorithm for computing $V(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\mathtt{F}\varphi])$:

From any state s where $r^r_{\text{state}}(s) < 0$ or $r^r_{\text{trans}}(s, \overline{a}) < 0$ for some action tuple $\overline{a}$, under all profiles of G, with probability 1 we reach either a state satisfying φ or a state where all rewards are zero and which cannot be left with probability 1 under all profiles.

We need to use value iteration twice to compute $V(\langle\langle C\rangle\rangle R^r_{\max=?}[\mathtt{F}\varphi])$. We start by finding the set of states for which the reward is infinite S^ρ_∞. This contains exactly the states satisfying $\langle\langle C\rangle\rangle P_{<1}[\mathtt{F}\varphi]$. At these states, C can try to avoid reaching φ forever, and since the probability of reaching φ is less than 1, there is a non-zero probability of never reaching φ, resulting in an infinite expected reward. This can be found as the complement of the set of states with $\langle\langle C\rangle\rangle P_{\min=1}[\mathtt{F}\varphi] = \langle\langle N \setminus C\rangle\rangle P_{\max=1}[\mathtt{F}\varphi]$ which we can use the pre-computation algorithm for the unbounded until probabilistic property to find.

We then select a $\gamma > 0$ and construct $r_\gamma = (r^\gamma_{\text{state}}, r^\gamma_{\text{trans}})$ with

$$r^\gamma_{\text{state}}(s) = \begin{cases} r^r_{\text{state}}(s) & \text{if } r^r_{\text{state}}(s) \neq 0 \\ \gamma & \text{otherwise} \end{cases}$$

$$r^\gamma_{\text{trans}}(s, \vec{a}) = \begin{cases} r^r_{\text{trans}}(s, \vec{a}) & \text{if } r^r_{\text{trans}}(s, \vec{a}) \neq 0 \\ \gamma & \text{otherwise} \end{cases}$$

and apply value iteration to approximate $V\Big(\langle\langle C\rangle\rangle R^{r_\gamma}_{\max=?}[\mathtt{F}\varphi]\Big)$ via

$$V\Big(\langle\langle C\rangle\rangle R^{r_\gamma}_{\max=?}[\mathtt{F}\varphi]\Big)(s) = \lim_{n\to\infty} v^s_n$$

$$v^s_n = \begin{cases} 0 & \text{if } s \in \text{Sat}(\varphi) \\ \infty & \text{if } s \in S^\rho_\infty \\ 0 & \text{if } n = 0 \\ \text{val}(Z(s)^n) + r^\gamma_{\text{state}}(s) & \text{otherwise} \end{cases}$$

$$Z(s)^n_{i,j} = r^\gamma_{\text{trans}}\big(s, (a_i, b_j)\big) + \sum_{s' \in S} \delta^C\big(s, (a_i, b_j)\big)(s') \times v^{s'}_{n-1}$$

We then perform a second value iteration with the original reward structure r but with initial values from the previous value iteration to get the final values of $V(\langle\langle C\rangle\rangle R^r_{\max=?}[\mathtt{F}\varphi])$.

$$V(\langle\langle C\rangle\rangle R^r_{\max=?}[\mathtt{F}\varphi])(s) = \lim_{n\to\infty} u^s_n$$

$$u^s_n = \begin{cases} 0 & \text{if } s \in \text{Sat}(\varphi) \\ \infty & \text{if } s \in S^\rho_\infty \\ V\Big(\langle\langle C\rangle\rangle R^{r_\gamma}_{\max=?}[\mathtt{F}\varphi]\Big)(s) & \text{if } n = 0 \\ \text{val}(Z(s)^n) + r^r_{\text{state}}(s) & \text{otherwise} \end{cases}$$

$$Z(s)^n_{i,j} = r^r_{\text{trans}}\big(s, (a_i, b_j)\big) + \sum_{s' \in S} \delta^C\big(s, (a_i, b_j)\big)(s') \times u^{s'}_{n-1}$$

3.4. Algebraic Decision Diagrams

Symbolic probabilistic model checking uses decision diagrams and their operations to efficiently perform formal verification. As probabilities and rewards are real-valued, we use ADDs [15] instead of Binary Decision Diagrams (BDDs) [31], with the terminals having values from $\mathbb{R}$ rather than $\mathbb{B} \equiv \{0, 1\}$, to represent functions of the form $f : \mathbb{B}^n \to \mathbb{R}$.

Many standard BDD operations have been generalised to ADDs. We summarise the operations used below:

$\text{Apply}(\text{op}, B_1, B_2)$ returns an ADD representing the function $f(s) = B_1(s)$ op $B_2(s)$. Here, op is a binary operator over the terminal values of the ADDs. Note that x op y need not lie in the same domain as x or y. We commonly use $\text{op} \in \{<, \leq, =, \neq, \geq, >\}$. We often write $\text{Apply}(\text{op}, B_1, B_2)$ with infix notation: B_1 op B_2 or as a Boolean function $\text{op}(B_1, B_2)$. We use non-standard conventions: $0 \div 0 = 0 \times \infty = 0 \times -\infty = 0$ for $\text{Apply}(\div, B_1, B_2)$ and $\text{Apply}(\times, B_1, B_2)$, which are useful for normalising and masking.

$\text{Abstract}(\text{op}, B, \overline{x})$ returns an ADD that represents the function $f(s) = \text{OP}_{\overline{x}}\, B(\langle s, \overline{x}\rangle)$. E.g. if op is max, then $f(s) = \max_{\overline{x}} B(\langle s, \overline{x}\rangle)$. Here, $\overline{x} \subseteq \text{vars}(B)$ and op is a commutative and associative binary operator over the terminal values of the ADD. For notational convenience, we define $\text{ThereExists}(\overline{x}, B) \coloneqq \text{Abstract}(\vee, B, \overline{x})$ and $\text{ForAll}(\overline{x}, B) \coloneqq \text{Abstract}(\wedge, B, \overline{x})$.

$\text{ReplaceVars}(\overline{x} \mapsto \overline{y}, B)$ returns an ADD with variables in $\overline{x}$ replaced with their corresponding variables in $\overline{y}$ in B. Here, $\overline{x}$ and $\overline{y}$ are of the same length.

$\text{Constant}(r)$ creates an ADD that represents the constant function $f(s) = r$ for all s.

$\text{IfThenElse}(B_{\text{cond}}, B_{\text{true}}, B_{\text{false}})$ takes a BDD B_{cond} and two ADDs B_{true} and B_{false} and returns the ADD that represents the function

$$f(s) = \begin{cases} B_{\text{true}}(s) \text{ if } B_{\text{cond}}(s) = 1 \\ B_{\text{false}}(s) \text{ otherwise} \end{cases}$$

$\text{Restrict}(B, B_{\text{care}})$ takes an ADD B and a BDD B_{care} and returns an ADD that represents the function

$$f(s) = \begin{cases} B(s) & \text{if } B_{\text{care}}(s) = 1 \\ \text{anything} & \text{otherwise} \end{cases}$$

This is used to reduce the size of B while maintaining correctness within the domain we care about. Its implementation is described in [18].

$\text{MVMult}(T, B, \overline{x}) \equiv \text{Abstract}(+, T \times B, \overline{x})$. Here, $T : \overline{x} \times \overline{y} \to \mathcal{D}$ represents a matrix with elements in $\mathcal{D}$, $B : \overline{x} \to \mathcal{D}$ represents a vector with elements in $\mathcal{D}$. $\text{MVMult}(T, B, \overline{x}) : \overline{y} \to \mathcal{D}$ represents the result of multiplying matrix T with vector B. This is implemented via the matrix-matrix multiplication algorithm in [32].

3.5. PRISM-games

PRISM-games [5] is an extension of the PRISM model checker [33] that supports modelling and verification of several game models, including CSGs. We use the PRISM-games modelling language for CSGs in this thesis, with our implementations extending PRISM-games with symbolic CSG verification.

It currently implements an explicit-state CSG model checker that uses arrays and bitsets to represent vectors and sets. We use this as an oracle to test our symbolic implementations and as a comparison point for benchmarking. Since caching previously solved matrix games can improve performance, we also implement a cached version of the explicit-state model checker to provide a more competitive comparison in Chapter 8.

Since there is existing code to lex and parse PRISM-games CSG models, when constructing symbolic models, we can directly take as input the abstract syntax tree of the models.

3.5.1. PRISM Language for CSGs

We target CSG models that can be represented with the following grammar:

```
program         := "csg" decl*
decl            := const_decl | player_decl | module_decl | reward_decl
const_decl      := "const" const_type var_name ("=" init_value)? ";"
player_decl     := "player" player_name mod_name ("," mod_name)* "endplayer"
module_decl     := "module" mod_name var_decl* command* "endmodule"
reward_decl     := "rewards" rew_struct_name reward_element* "endrewards"

const_type      := "int" | "double" | "bool"
var_decl        := var_name ":" var_type ("init" init_value)? ";"
var_type        := "bool" | "[" int_literal ".." int_literal "]"

command         := "[" action_list "]" guard "->" outcome ("+" outcome)* ";"
action_list     := "" | action ("," action)*
outcome         := (prob ":")? update

guard           := boolean_expr
update          := (update_element ("&" update_element)*)
update_element  := var_name "'" "=" expr
prob            := double_expr

reward_element  := state_reward | trans_reward
state_reward    := guard ":" expr ";"
trans_reward    := "[" action_list "]" guard ":" expr ";"
```

Listing 1: PRISM CSG Grammar

The PRISM-games CSG language also allows additional syntactic sugar for renaming of variables to easily generate similar modules, type inference of constants, etc. We will not discuss these here as such programs can be rewritten in the above grammar in a straightforward manner. We also do not deal with global variables since it is possible to rewrite CSG models with global variables to put the global variables into a non-player module.

For example, Listing 2 is a CSG model of two processes trying to transmit over a shared channel, adapted from [34].

```
csg

player p1 mac1 endplayer
player p2 mac2 endplayer

const int emax; // energy bound
const double q1; // probability single user successfully transmits
const double q2; // probability two users successfully transmit

module channel // used to compute joint probability distribution for
transmission failure
  c : bool init false; // is there a collision?
  [t1,w2] true -> q1 : (c'=false) + (1-q1) : (c'=true); // only user 1 transmits
  [w1,t2] true -> q1 : (c'=false) + (1-q1) : (c'=true); // only user 2 transmits
  [t1,t2] true -> q2 : (c'=false) + (1-q2) : (c'=true); // both users transmit
endmodule

module mac1 // user 1
  s1 : [0..1] init 0; // has player 1 sent?
  e1 : [0..emax] init emax; // energy level of player 1
  [w1] true -> (s1'=0); // wait
  [t1] e1>0 -> (s1'=c'?0:1) & (e1'=e1-1); // transmit
endmodule

// construct second user with renaming
module mac2 = mac1 [ s1=s2, e1=e2, w1=w2, t1=t2 ] endmodule

// reward structure for player 1
rewards "r1"
  s1=1 : 1;
endrewards

// reward structure for player 2
rewards "r2"
  s2=1 : 1;
endrewards
```

Listing 2: PRISM-games CSG Example: Medium Access Control

In CSG models, modules are either player modules or non-player modules, depending on whether they are controlled by a player. At each time step, each player chooses an action that is executable in all of their modules. Non-player modules serve to react to player actions, with their command selected based on the action tuple selected. All modules then execute simultaneously, resulting in a probabilistic transition to the next state. Variables

are either primed or non-primed, where primed variables represent the next state of the variable after a command is executed.

We say a command is valid with respect to a state if its guard is satisfied by that state. We say a command is active with respect to a state and action tuple if it is valid with respect to that state and its action(s) are a subset of the action tuple.

For player modules, each command's `action_list` must contain at least one action, and the first action must be controlled by that player. All other actions are from different players. For any state, consider all valid commands. If any commands have action a as the first action in the action list, then these commands must form a complete partition over the other actions that can be taken by the other players. This ensures that whenever action a is available to the player, the resulting behaviour is defined regardless of the actions taken by the other players.

For non-player modules, each command's action list similarly can contain multiple actions, all from different players, but is also allowed to be empty. For the model to be well-defined, there can be at most one active command for any given state and action tuple combination. If there are zero active commands, the module simply remains in the same state.

Within a single command, the probabilities of all outcomes must sum to 1. This ensures that the probability distribution over next states is well-defined. The `guard` must only consist of non-primed variables and can be over any variables, including those from other modules. The `expr` may contain both non-primed and primed variables, including variables from other modules. `update_element`s within an `update` must update distinct primed variables belonging to the module in which the command is defined.

Allowing primed variables as part of the `expr` lets commands depend on the next state of other variables, which is useful for defining coordinated transitions. In the above example, it ensures both `p1` and `p2` can update their `s` variables consistently with whether `c` is updated to true or false in the `channel` module. This is a powerful feature unique to CSG models in PRISM-games, as related modelling languages such as the PRISM language for MDPs or TSGs do not allow primed variables in updates. Note that for the next state to be well-defined, there must be no cyclic dependencies between primed variables across commands in different modules.

These restrictions ensure that for every state and action tuple, at most one command per module is active, and that the resulting probability distributions are well-defined.

The PRISM-games CSG modelling language implies rectangular availability: an action tuple is available exactly when each player's selected action is available. This property is used later when solving matrix games in the model checking algorithms.

3.6. Two-Player Zero-Sum Games in Normal Form

A two-player zero-sum game in normal form can be represented by a payoff matrix $M \in \mathbb{R}^{n \times m}$ where player A has n actions and player B has m actions. The entry $M(i, j)$ represents the utility that player A receives when players A and B play actions i and j respectively. Since this is a zero-sum game, player B receives utility $-M(i, j)$.

Solving a two-player zero-sum game involves finding a Nash equilibrium, which is a pair of mixed strategies $(x_A \in \mathbb{R}^n, x_B \in \mathbb{R}^m)$ for players A and B such that neither player can improve their expected utility by unilaterally changing their strategy.

This is equivalent to solving the minimax problem:

$$\max_{x_A \in \Delta_n} \min_{x_B \in \Delta_m} x_A^\top M x_B$$

where x_A and x_B are probability distributions over the action sets of players A and B respectively, i.e. $\Delta_k = \left\{ x \in \mathbb{R}^k \mid \sum_{i=1}^k x_i = 1, x_i \geq 0 \right\}$.

The strategies x_A and x_B can be found using various algorithms, including linear programming, the Lemke-Howson algorithm [35], and iterative methods like fictitious play and regret matching [36].

The existing explicit-state model checking algorithm for CSGs relies on linear programming to solve matrix games, which we will leverage in our hybrid implementation in Chapter 7. Since linear programming is difficult to translate into a symbolic form, we instead focus on regret matching for our fully symbolic approach.

3.7. (Discounted) Regret Matching

Regret matching [36] and its variants are a class of algorithms that can be used to compute correlated equilibria [37] in normal form games. For two-player zero-sum games, all correlated equilibria yield the same value as the minimax value (the unique Nash equilibrium value) [38]; thus, we can use regret matching to compute the minimax value in these games.

There are many variations of regret matching, including the original regret matching, regret matching+ [39], discounted regret matching [40], and several more recent advanced

variants. We focus on discounted regret matching as it has been shown to have similar or better convergence properties compared to other classic variants in practice.

Regret matching algorithms work iteratively to produce a series of strategies for each player, based on playing actions that have high regret (of not being played) in the past. The (weighted) average of these strategies over time converges to a correlated equilibrium.

Here, we describe the discounted regret matching algorithm in a detailed form that is suitable for use on two-player matrix games and can be translated into a symbolic version (see Chapter 6).

Suppose we are solving a single two-player zero-sum game

$$\max_{x_A \in \Delta_n} \min_{x_B \in \Delta_m} x_A^\top M x_B$$

with $M \in \mathbb{R}^{n \times m}$, $x_A \in \mathbb{R}^n$, and $x_B \in \mathbb{R}^m$. The algorithm works as follows:

Initialisation:
0.

We select discounting parameters $\alpha, \beta, \gamma \in \mathbb{R}$ that control how quickly past regrets and strategies are discounted.

In particular, the original paper recommends using $\alpha = 1.5, \beta = 0, \gamma = 2$.

1.

We initialise a uniform strategy for each player $x_A^0 \in [0,1]^n$ and $x_B^0 \in [0,1]^m$ over their respective action sets:

$$x_A^0 \in [0,1]^n = \frac{1}{n}\mathbf{1}$$

$$x_B^0 \in [0,1]^m = \frac{1}{m}\mathbf{1}$$

2.

We initialise the cumulative discounted regret for each action for each player to be 0:

$$r_A^0 \in \mathbb{R}^n = \mathbf{0}$$

$$r_B^0 \in \mathbb{R}^m = \mathbf{0}$$

3.

We initialise a vector to keep track of the unnormalised average strategy for each player:

$$\sigma_A^0 \in \mathbb{R}^n = \mathbf{0}$$
$$\sigma_B^0 \in \mathbb{R}^m = \mathbf{0}$$

For each iteration $t \geq 1$:

1. Compute the utility vector for each player based on the opponent's previous strategy:

$$u_A^t \in \mathbb{R}^n = M x_B^{t-1}$$
$$u_B^t \in \mathbb{R}^m = -M^\top x_A^{t-1}$$

This utility vector gives the payoff of each pure action against the opponent's previous mixed strategy.

2. Compute the payoff value for each player $p \in \{A, B\}$ assuming they each play their previous strategy:

$$c_p^t \in \mathbb{R} = \left(x_p^{t-1}\right)^\top u_p^t$$

3. Compute the instantaneous regret for each action for each player:

$$i_A^t \in \mathbb{R}^n = u_A^t - \mathbf{1}c_A^t$$
$$i_B^t \in \mathbb{R}^m = u_B^t - \mathbf{1}c_B^t$$

The instantaneous regret for an action is the difference between the utility of that action and the expected utility given the player's previous strategy. That is, we have more positive regret for actions that would have been better to play.

4. Discount and accumulate the regret for each action for each player:

$$r_A^t(j) \in \mathbb{R} = \begin{cases} \text{if } r_A^{t-1}(j) \geq 0 \text{ then } & \frac{t^\alpha}{1+t^\alpha} r_A^{t-1}(j) + i_A^t(j) \\ \text{else} & \frac{t^\beta}{1+t^\beta} r_A^{t-1}(j) + i_A^t(j) \end{cases} \text{for } j \in 1..n$$

$$r_B^t(j) \in \mathbb{R} = \begin{cases} \text{if } r_B^{t-1}(j) \geq 0 \text{ then } & \frac{t^\alpha}{1+t^\alpha} r_B^{t-1}(j) + i_B^t(j) \\ \text{else} & \frac{t^\beta}{1+t^\beta} r_B^{t-1}(j) + i_B^t(j) \end{cases} \text{for } j \in 1..m$$

5. Compute the new strategy for each player by normalising the positive cumulative regret:

$$y_A^t(j) \in \mathbb{R} = \begin{cases} \text{if } r_A^t(j) \geq 0 \text{ then } r_A^t(j) \\ \text{else} \qquad\qquad\quad 0 \end{cases} \text{ for } j \in 1..n$$

$$x_A^t \in [0,1]^n = \begin{cases} \text{if} \sum_{j=1}^n y_A^t(j) > 0 \text{ then } y_A^t \div \sum_{j=1}^n y_A^t(j) \\ \text{else} \qquad\qquad\qquad\quad x_A^0 \end{cases}$$

$$y_B^t(j) \in \mathbb{R} = \begin{cases} \text{if } r_B^t(j) \geq 0 \text{ then } r_B^t(j) \\ \text{else} \qquad\qquad\quad 0 \end{cases} \text{ for } j \in 1..m$$

$$x_B^t \in [0,1]^m = \begin{cases} \text{if} \sum_{j=1}^m y_B^t(j) > 0 \text{ then } y_B^t \div \sum_{j=1}^m y_B^t(j) \\ \text{else} \qquad\qquad\qquad\quad x_B^0 \end{cases}$$

6. Accumulate the current strategy for each player:

$$\sigma_A^t \in \mathbb{R}^n = \left(\frac{t}{t+1}\right)^\gamma \sigma_A^{t-1} + x_A^t$$

$$\sigma_B^t \in \mathbb{R}^m = \left(\frac{t}{t+1}\right)^\gamma \sigma_B^{t-1} + x_B^t$$

7. Compute the average strategy for each player by normalising the accumulated strategy[2]:

$$\overline{x}_A^t \in [0,1]^n = \sigma_A^t \div \sum_{j=1}^n \sigma_A^t(j)$$

$$\overline{x}_B^t \in [0,1]^m = \sigma_B^t \div \sum_{j=1}^m \sigma_B^t(j)$$

8. Check for termination:

We can compute an upper bound for the minimax value at iteration t

$$\max_{x_A} (x_A)^\top M(\overline{x}_B^t)$$

and a lower bound as

$$\min_{x_B} (\overline{x}_A^t)^\top M(x_B)$$

[2]After the first iteration, the accumulated strategy should never be 0, so this case need not be checked.

We can therefore terminate when the difference between the upper bound and lower bound is less than some threshold $\varepsilon > 0$.

9. When we terminate, we return the final result:

$$\text{val}(M) = (\overline{x}_A^t)^\top M(\overline{x}_B^t)$$

Chapter 4

Symbolic Representation and Construction of CSGs

4.1. Symbolic Model Representation

We aim to construct an ADD T and a BDD Avail that represent the transition function δ and the availability set Δ from Section 3.1:

$$T : S \times \prod_{p \in N} \mathrm{Act}_p \times S' \to [0, 1]$$

$$\mathrm{Avail} : S \times \prod_{p \in N} \mathrm{Act}_p \to \mathbb{B}$$

where Act_p denotes the ADD variables used to encode the action chosen by player p, including the encoding of $\bot_p$. If $\mathrm{Avail}(s, \vec{a}) = 0$, then we require $T(s, \vec{a}, s') = 0$ for all s'.

For each reward structure, we want to construct ADDs that represent the two reward functions:

$$R^r_{\mathrm{state}} : S \to \mathbb{R}$$

$$R^r_{\mathrm{trans}} : S \times \prod_{p \in N} \mathrm{Act}_p \to \mathbb{R}$$

We first construct $T(s, \vec{a}, s')$, $R^r_{\mathrm{state}}(s)$ and $R^r_{\mathrm{trans}}(s, \vec{a})$ to take arbitrary values when s or s' do not encode legitimate states. These illegitimate values will be zeroed out when we filter by reachability.

4.2. ADD Variables

Before constructing the ADDs, we declare the following groups of ADD variables.

Group	Symbol	Number of Variables
Current state variables	S	For each `int`: $\lceil \log_2(\text{range}(x)) \rceil$ variables For each `bool`: 1 variable
Next state variables	S'	Same as above
Action variables	$\bigcup_{p \in N} \text{Act}_p$	$\sum_{p \in N} \lceil \log_2(n_p) \rceil$ variables where n_p is the number of actions for player p, including $\bot_p$

Table 1: Groups of ADD Variables

Variable ordering is important for keeping ADDs compact and efficient.

We use the typical interleaved ordering for current-state and next-state variables that is used in PRISM-games for symbolic representation of MDPs and TSGs [13].

For action variables, we can decide to order them before, interleaved with, or after the state variables. However, to make the hybrid solver in Chapter 7 work, we will need to order all action variables after all state variables.

4.2.1. Dynamic Variable Reordering

To further improve the ordering, we enable CUDD's automatic dynamic reordering. During model construction, we use the window-permutation method with window size 2 [41, 42] as it performed best in ad hoc testing. Since parts of PRISM-games assume the original order is preserved, we restore that order at the end of model construction. This also restores the action variables to positions after the state variables for the hybrid solver.

4.3. Construction of Transition Relation

To construct the transition ADD, T, we translate each command in each module separately, compose them together to form ADDs for each module, then finally compose the module ADDs together to form T.

We first focus on translating a single CSG command. In general, the ith command of module M has the form

$$\kappa_i^M := [\alpha_{i,1}^M, \alpha_{i,2}^M, ..., \alpha_{i,h}^M] \quad \gamma_i^M \to \lambda_{i,1}^M : \mu_{i,1}^M + ... + \lambda_{i,m}^M : \mu_{i,m}^M$$

We drop the M superscript when only talking about a single module.

Here, $\mu_{i,j}$ corresponds to an `update` in the grammar while $\lambda_{i,j}$ is an update probability dependent on the current state. We observe that $\mu_{i,j}$ is a relation over the old and new state variables. We construct a BDD $\text{M}_{i,j}$ such that $\text{M}_{i,j}(s, s') = 1$ iff $\mu_{i,j}(s, s')$ holds.

Each $\mu_{i,j}$ has the form:

$$\mu_{i,j} = \xi_{i,j,1} \wedge ... \wedge \xi_{i,j,q}$$

where $\xi_{i,j,k} = \big(x_{i,j,k} = e_{i,j,k}\big)$ corresponds to a single `update_element`, with $x_{i,j,k}$ being a primed variable, and $e_{i,j,k}$ being an expression over primed and unprimed variables. We will translate each of these `update_elements` into BDDs over S and S' individually.

To do so, we construct ADDs $L_{i,j,k} : S \times S' \to \mathbb{Z}$ and $R_{i,j,k} : S \times S' \to \mathbb{Z}$ that represent the left-hand side and right-hand side of each update element respectively. These are constructed recursively over their respective expressions, with the recursive cases being identical to those used in symbolic model checking of other models in PRISM [13]. The elegant trick permitting the use of primed variables in updates is allowing base cases where the variable is primed; we simply return a BDD f where $f(s, s')$ represents the value of the primed variable in state s'. We can then set $\Xi_{i,j,k} = \text{Apply}\big(=, L_{i,j,k}, R_{i,j,k}\big)$.

These are joined together to form an ADD representing the entire update.

$$\text{M}_{i,j} = \Xi_{i,j,1} \wedge ... \wedge \Xi_{i,j,q}$$

We may also need to modify $\mu_{i,j}$ from what is seen in the CSG model's description to include constant update elements of the form $v' = v$ for all variables v that are owned by the module but not changed by the command. Otherwise, there will be additional degrees of freedom that allow unintended transitions in the overall transition ADD. We can handle these unchanged variables while only considering each command separately, since in each module only one command will be the active command for any state and action tuple, and furthermore, variables can only be modified by the module that owns them.

We then construct a BDD A to encode the action list, such that

$$\text{A}_i(a_1, ..., a_h) = \begin{cases} 1 \text{ if } a_{\alpha_j} = \alpha_j \forall j \quad \in [1..h] \\ 0 \text{ otherwise} \end{cases}$$

where a_{α_j} is the action selected by its player p_{α_j}.

We do so by creating BDDs $\text{A}_{i,j} : \text{Act}_{p_{\alpha_j}} \to \mathbb{B}$ that represent whether action α_j is selected by p_{α_j}. In our implementation, actions of player p are binary encoded by Act_p, thus we order all the actions of each player and $\text{A}_{i,j}$ is constructed based on the binary representation of α_j's index amongst the actions of p_{α_j}.

We next set $\mathrm{A}_i = \bigwedge_{j \in [1..h]} \mathrm{A}_{i,j}$. The action variables for players with none of their actions mentioned in the action list do not affect the validity of the command, which is the intended semantics. For an empty action list, A_i is the constant true BDD.

We construct BDD Γ_i to encode the guard, such that $\Gamma_i(s) = 1$ iff $\gamma_i(s)$ holds. This is done by recursing over the structure of the guard, and is the same as what is done in symbolic model checking of MDPs and TSGs.

Similarly, we construct $\Lambda_{i,j}$ to encode the update probability $\lambda_{i,j}$ by recursing over the structure of $\lambda_{i,j}$, with the base case being a constant probability.

We combine these together for the whole command as

$$\mathrm{K}_i = (\mathrm{A}_i \wedge \Gamma_i) \times \left(\Lambda_{i,1} \times \mathrm{M}_{i,1} + ... + \Lambda_{i,m} \times \mathrm{M}_{i,m}\right)$$

To translate whole modules, we combine the effects of ADDs within the same module on the module's state by adding them together, $\Theta^M = \mathrm{K}_1^M + ... + \mathrm{K}_c^M$. This is valid as all $\mathrm{A}_i \wedge \Gamma_i$ are disjoint and so only one command can be active for any state and action tuple.

For non-player modules M, we also create an ADD representing the identity transition for the case where no command is active:

$$\mathrm{K}_{\mathrm{id}}^M = \neg\left(\left(\mathrm{A}_1^M \wedge \Gamma_1^M\right) \vee ... \vee \left(\mathrm{A}_c^M \wedge \Gamma_c^M\right)\right) \times \mathrm{M}_{\mathrm{id}}^M = \neg \mathrm{Z}_{\mathrm{active}}^M \times \mathrm{M}_{\mathrm{id}}^M$$

where $\mathrm{M}_{\mathrm{id}}^M$ is the identity relation over the state variables of module M and $\mathrm{Z}_{\mathrm{active}}^M$ is the BDD representing whether any command in module M is active with respect to the current state and action tuple.

For each non-player module, we add its identity transition for states/action tuples where no command is active, and then multiply the resulting module ADDs because modules execute synchronously.

$$T_{\text{non-player}} = \prod_{\text{non-player module } M} \left(\Theta^M + \mathrm{K}_{\mathrm{id}}^M\right)$$

For player modules, we cannot construct an idle action only based on the guards of commands within the module. The condition for the idle action being available is that every non-idle action is unavailable in at least one of the player's modules. We thus need to

consider all modules of a specific player before being able to construct the idle transitions for the player's modules.

This means that for each player p and action a of p, we should export from each module M, a BDD $\text{Available}_{p,a}^{M}$ that represents whether action a is available in module M. An action a is available in module M if there exists one command i with $\alpha_{i,1}^{M} = a$ such that γ_i^M is satisfied by the current state. Furthermore, we also need to export the identity transition M_{id}^{M} for each module M controlled by player p.

We can then join these separately to form the identity update for all the player's modules: $\prod_{M \in \text{Modules}_p} \text{M}_{\text{id}}^{M}$, as well as the condition for the idle action being available:

$$\text{Stuck}_p = \prod_{a \in A_p \setminus \left\{a_{\perp_p}\right\}} \text{Stuck}_{p,a} = \prod_{a \in A_p \setminus \left\{a_{\perp_p}\right\}} \neg \prod_{M \in \text{Modules}_p} \text{Available}_{p,a}^{M}$$

We then combine this together with the rest of the module transitions for player p via

$$T_p = \left(\text{Stuck}_p \times \text{A}_{\perp_p} \times \prod_{M \in \text{Modules}_p} \text{M}_{\text{id}}^{M} \right) + \prod_{M \in \text{Modules}_p} \Theta^M$$

The overall transition relation is then given by

$$T = T_{\text{non-player}} \times \prod_p T_p$$

4.4. Correctness Argument

We now sketch why the construction is correct.

A_i^M is a BDD that returns true if and only if the action tuple $\overline{a}$ is compatible with the action list of command κ_i^M. Γ_i^M is a BDD that returns true if and only if the guard of command κ_i^M is satisfied by state s. $\text{M}_{i,j}^{M}$ is a BDD that returns true if and only if the update $\mu_{i,j}^{M}$ holds between states s and s'.

Thus, K_i^M is an ADD such that $\text{K}_i^M(s, \overline{a}, s')$ gives the probability that we transition from s to s' when action tuple $\overline{a}$ is taken, assuming that M only contains the single command κ_i^M and that all variables from other modules can take arbitrary values. If s does not satisfy the guard, or $\overline{a}$ does not satisfy the action list, then this probability will correctly be 0 for every s'. On the other hand, if s satisfies the guard and $\overline{a}$ satisfies the action list, then for every s' that respects some update(s) of κ_i^M, the probability will be the sum of the relevant $\lambda_{i,j}^{M}$ of the update(s) that result in s' for the variables of module M.

Θ^M is then an ADD such that $\Theta^M(s, \overline{a}, s')$ gives the probability that we transition from s to s' when action tuple $\overline{a}$ is taken, assuming that M contains all its commands, all variables from other modules can take arbitrary values and there are no idle transitions for module M. This comes from the fact that for every $(s, \overline{a}, s')$, $\Theta^M(s, \overline{a}, s') = \mathrm{K}_i^M(s, \overline{a}, s')$ for the unique command κ_i^M that is active for $(s, \overline{a})$, and 0 if no command is active.

For non-player modules, $\Theta^M + \mathrm{K}_{\mathrm{id}}^M$ is such that $\left(\Theta^M + \mathrm{K}_{\mathrm{id}}^M\right)(s, \overline{a}, s')$ gives the probability that we transition from s to s' when action tuple $\overline{a}$ is taken, assuming that M contains all its commands, all variables from other modules can take arbitrary values and idle transitions are allowed. The only difference from Θ^M is that if no command is active for $(s, \overline{a})$, then we transition to s' with probability 1 if $s = s'$, and 0 otherwise, which is the intended semantics.

For player modules, $\prod_{M \in \mathrm{Modules}_p} \Theta^M$ gives the probability that we transition from s to s' when action tuple $\overline{a}$ is taken, assuming that all modules of player p contain all their commands, all variables from modules not owned by p can take arbitrary values and no idle transitions are allowed by player p. This is because the relevant update that is selected by each module that respects the transition $(s, \overline{a}, s')$ is done independently, and so we can multiply the probabilities together. For states s where no non-idle action can be taken, $\left(\prod_{M \in \mathrm{Modules}_p} \Theta^M\right)(s, \overline{a}, s') = 0$ for each $\overline{a}$ and s', allowing us to add on idle transitions.

T_p then includes the idle transition for player p. $T_p(s, \overline{a}, s')$ with $a_p = \bot_p$ has a value of 1 if and only if s and s' agree on the values within modules owned by p and p is stuck. Apart from where $a_p = \bot_p$, it has the same behaviour as $\prod_{M \in \mathrm{Modules}_p} \Theta^M$, thus it correctly modifies the transition probabilities only for player p's idle action.

Finally, since all modules execute synchronously and the relevant updates are selected independently of each other, the overall transition ADD T correctly gives the probability of transitioning from s to s' when action tuple $\overline{a}$ is taken, accounting for idle actions from all players and identity transitions of non-player modules.

4.5. Construction of Reward Structures

A state reward item has the form:

$$\upsilon_i = \gamma_i : \rho_i$$

A transition reward item has the form:

$$\iota_i = \left[\alpha_{i,1}, \alpha_{i,2}, ..., \alpha_{i,h}\right]\ \gamma_i : \rho_i$$

where $\alpha_{i,j}$ are actions, all from different players, γ_i is a guard, and ρ_i is a reward expression.

Similar to the translation of each command, we translate each reward item separately and merge them together.

Suppose there are q state reward items and m transition reward items. For each state reward item υ_i, we construct an ADD to represent the reward for being in specific states as

$$\Upsilon_i = \Gamma_i \times \mathrm{P}_i$$

For each transition reward item ι_i, we construct an ADD to represent the reward for taking specific transitions as

$$\mathrm{I}_i = (\Gamma_i \wedge \mathrm{A}_i) \times \mathrm{P}_i$$

where A_i and Γ_i are constructed similarly as in the command translation. P_i is constructed recursively over the structure of the reward expression in a manner identical to existing work done for symbolic model checking of TSGs [14].

Since all reward elements independently contribute to the overall reward as part of a sum, we can combine them via:

$$R_{\text{state}} = \sum_{i=1}^{q} \Upsilon_i$$

$$R_{\text{trans}} = \sum_{i=1}^{m} \mathrm{I}_i$$

4.6. Reachability

Reachability restricts the set of states considered when the model checking algorithm is run from a set of initial states S_{initial}. This prunes illegitimate and unreachable states from the ADDs.

The set of reachable states can be computed symbolically using a fixed point computation. We first define the deterministic transition relation T_{det} to be:

$$T_{\text{det}} : S \times S' \to \mathbb{B} = \text{ThereExists}\left(\prod_{p \in N} \text{Act}_p, T > \text{Constant}(0)\right)$$

This gives us $T_{\text{det}}(s, s') = 1$ iff there is some action tuple $\overline{a}$ such that $T(s, \overline{a}, s')$ is positive, i.e. it is possible to directly transition from s to s'.

We then use the standard reachability least fixed-point computation from non-probabilistic symbolic model checking:

$$U^0_{\text{reachable}} = S_{\text{initial}}$$
$$U^k_{\text{reachable}} = U^{k-1}_{\text{reachable}} \vee \text{ReplaceVars}\big(S' \mapsto S, \text{ThereExists}\big(S, U^{k-1}_{\text{reachable}} \wedge T_{\text{det}}\big)\big)$$

We can then prune T to only contain transitions from reachable states via:

$$T_{\text{reachable}} = T \times U_{\text{reachable}}$$

and this $T_{\text{reachable}}$ can then be used for model checking.

We thus reassign $T \coloneqq T_{\text{reachable}}$.

We now compute the availability function:

$$\text{Avail} = \text{ThereExists}(S', T > 0)$$

and can prune the reward functions:

$$R'_{\text{state}} = R_{\text{state}} \times U_{\text{reachable}}$$
$$R'_{\text{trans}} = R_{\text{trans}} \times \text{Avail}$$

Chapter 5

Symbolic Model Checking of CSGs

We adapt the model checking algorithms in Section 3.3 to work symbolically with ADDs.

We will represent the solution to a property check $V_{G,C}(\varphi) : S \to \mathbb{B}$ from Section 3.3 as a BDD of the same form.

To compute $V(\varphi)$, we build up the solution recursively over the structure of φ.

We let $V_{\mathrm{U}} : S \to \mathbb{B}$ be the BDD representing the set of all reachable states from S_{initial}, computed in Section 4.6.

The simple cases are:

$$\begin{aligned} V(\text{true}) &= V_{\mathrm{U}} \\ V(a) &= V_{\mathrm{U}} \wedge S_a \text{ where } S_a \text{ is a BDD that represents } \{s \mid a \in L(s)\} \\ V(\neg\varphi) &= V_{\mathrm{U}} \wedge \neg V(\varphi) \\ V(\varphi_1 \wedge \varphi_2) &= V(\varphi_1) \wedge V(\varphi_2) \end{aligned}$$

For $\langle\!\langle C \rangle\!\rangle P_{\sim q}[\psi]$ we compute the following:

$$\begin{aligned} V\big(\langle\!\langle C \rangle\!\rangle P_{>q}[\psi]\big) &= V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle P_{\max=?}[\psi]) > \text{Constant}(q)) \\ V\big(\langle\!\langle C \rangle\!\rangle P_{\geq q}[\psi]\big) &= V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle P_{\max=?}[\psi]) \geq \text{Constant}(q)) \\ V\big(\langle\!\langle C \rangle\!\rangle P_{<q}[\psi]\big) &= V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle P_{\min=?}[\psi]) < \text{Constant}(q)) \\ V\big(\langle\!\langle C \rangle\!\rangle P_{\leq q}[\psi]\big) &= V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle P_{\min=?}[\psi]) \leq \text{Constant}(q)) \end{aligned}$$

where $V(\langle\!\langle C \rangle\!\rangle P_{\max=?}(\psi)) : S \to [0,1]$ is the solution to the maximum probabilistic reachability problem for the coalition C with respect to path property ψ, as introduced in Section 3.2.3.

For $\langle\!\langle C \rangle\!\rangle R^r_{\sim x}[\rho]$ we analogously do:

$$V(\langle\!\langle C \rangle\!\rangle R^r_{>x}[\rho]) = V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\rho]) > \mathrm{Constant}(x))$$
$$V(\langle\!\langle C \rangle\!\rangle R^r_{\geq x}[\rho]) = V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle R^r_{\max=?}[\rho]) \geq \mathrm{Constant}(x))$$
$$V(\langle\!\langle C \rangle\!\rangle R^r_{<x}[\rho]) = V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle R^r_{\min=?}[\rho]) < \mathrm{Constant}(x))$$
$$V(\langle\!\langle C \rangle\!\rangle R^r_{\leq x}[\rho]) = V_{\mathrm{U}} \wedge (V(\langle\!\langle C \rangle\!\rangle R^r_{\min=?}[\rho]) \leq \mathrm{Constant}(x))$$

As in Section 3.3, to handle $R^r_{\min/\max=?}$ and $P_{\min/\max=?}$ we will need to solve multiple matrix games during the model checking process.

We first define the action set for the coalitions as

$$\mathrm{Act}_C \coloneqq \prod_{i \in C} \mathrm{Act}_i,\ \mathrm{Act}_{N \setminus C} \coloneqq \prod_{i \in N \setminus C} \mathrm{Act}_i$$

We can then define the operation

$$\begin{aligned}\mathrm{Val} : &\left(S \to \mathrm{Act}_C \times \mathrm{Act}_{N\setminus C} \to \mathbb{R}\right) \\ &\times \left(S \to \mathrm{Act}_C \times \mathrm{Act}_{N\setminus C} \to \mathbb{B}\right) \\ &\times \mathrm{ADDVars} \times \mathrm{ADDVars} \to (S \to \mathbb{R})\end{aligned}$$

such that:

$\mathrm{Val}\big(Z, \mathrm{Av}, \mathrm{Act}_C, \mathrm{Act}_{N\setminus C}\big)(s)$ is the minimax value of the matrix game between C and $N \setminus C$ defined by $Z(s)$.

Z is the ADD that represents the payoff values for each state for each selected action tuple. We allow $Z\big(s, \vec{a}_C, \vec{a}_{N\setminus C}\big)$ to take arbitrary values when $\mathrm{Av}\big(s, \vec{a}_C, \vec{a}_{N\setminus C}\big) = 0$.

Av is the availability set that specifies which action tuples are admissible at each state, usually passed as Avail from Section 4.1.

Act_C and $\mathrm{Act}_{N\setminus C}$ specify which variables are part of the maximising and minimising players respectively.

This generalises the val operation used in Section 3.3 to solve multiple matrix games each indexed by states, simultaneously. We describe a purely symbolic algorithm to compute Val in Chapter 6 and an efficient hybrid algorithm in Chapter 7.

5.1. Maximum/Minimum Probabilistic Reachability

As $V(\langle\langle C\rangle\rangle P_{\min=?}(\psi)) = V(\langle\langle N \setminus C\rangle\rangle P_{\max=?}(\psi))$, we only need to describe how to compute $V(\langle\langle C\rangle\rangle P_{\max=?}(\psi))$.

As in the explicit-state algorithm, to compute $V(\langle\langle C\rangle\rangle P_{\max=?}(\psi))$, we consider the structure of ψ.

5.1.1. Next

If $\psi = \mathtt{X}\varphi$, we compute

$$V(\langle\langle C\rangle\rangle P_{\max=?}(\mathtt{X}\varphi)) = \mathrm{Val}\Big(\mathrm{MVMult}(T, \mathrm{ReplaceVars}(S \mapsto S', V(\varphi)), S'), \mathrm{Avail}, \mathrm{Act}_C, \mathrm{Act}_{N\setminus C}\Big)$$

5.1.2. Bounded Until

If $\psi = \varphi_1 \ \mathtt{U}^{\leq k} \ \varphi_2$, we compute $V_n = V(\langle\langle C\rangle\rangle P_{\max=?}(\varphi_1 \ \mathtt{U}^{\leq n} \ \varphi_2))$ for all $n \in [0..k]$ via:

$$\begin{aligned} V_0 &= V(\varphi_2) \\ V_n &= \mathrm{IfThenElse}\Big(V(\varphi_2), \mathrm{Constant}(1), \mathrm{Val}\Big(Z_n, \mathrm{Avail}_?, \mathrm{Act}_C, \mathrm{Act}_{N\setminus C}\Big)\Big) \\ \text{where } \mathrm{Avail}_? &= (V(\varphi_1) \wedge \neg V(\varphi_2)) \wedge \mathrm{Avail} \\ T_? &= \mathrm{Restrict}(\mathrm{Restrict}(T, \mathrm{Avail}_?), \mathrm{ReplaceVars}(S \mapsto S', V_{\mathrm{U}})) \\ Z_n &= \mathrm{MVMult}(T_?, \mathrm{ReplaceVars}(S \mapsto S', V_{n-1}), S') \end{aligned}$$

We make use of Restrict to reduce the size of T as much as possible while still preserving correctness on relevant states.

5.1.3. Until

Using the same logic as in Section 3.3, $V(\langle\langle C\rangle\rangle P_{\max=?}(\varphi_1 \ \mathtt{U} \ \varphi_2))$ can be found by doing the bounded until iteratively until convergence.

We describe in detail how to perform the pre-computation algorithms from [24] symbolically.

5.1.3.1. Prob 1

The set of states that have maximum probability 1 of satisfying $\varphi_1 \ \mathtt{U} \ \varphi_2$, $v(\langle\langle C\rangle\rangle P_{\max=1}(\varphi_1 \ \mathtt{U} \ \varphi_2)) \subseteq S$ can be found via an adaptation of the "almost-surely" reachable algorithm described in [24].

$$v(\langle\langle C\rangle\rangle P_{\max=1}(\varphi_1 \ \mathtt{U} \ \varphi_2)) = \nu y.\mu x.((\mathrm{apre}_C(y, x) \cap v(\varphi_1)) \cup v(\varphi_2))$$

where:

- $\text{apre}_C : 2^S \times 2^S \to 2^S$ is the predecessor operator for computing almost surely reachability. $\text{apre}_C(y, x)$ represents the states where the players in coalition C can ensure that the next state is in y with probability 1, and also ensure that there is a non-zero probability of reaching a state in x.
- ν and μ are the greatest and least fixed point operators respectively.
- $v(\varphi_1) \subseteq S$ and $v(\varphi_2) \subseteq S$ are the sets of states satisfying φ_1 and φ_2 respectively.

$\text{apre}_C(y, x)$ can be computed as follows:

$$
\begin{aligned}
s \in \text{apre}_C(y, x) \Leftrightarrow B_x^s\big(A_y^s\big) &= \text{act}_{N\setminus C}^s \\
A_y^s &= \Big\{a \in \text{act}_C^s \mid \forall b \in \text{act}_{N\setminus C}^s \ \forall s' \in S \ \delta(\langle\!\langle a, b\rangle\!\rangle, s, s') > 0 \to s' \in y\Big\} \\
B_x^s(A) &= \Big\{b \in \text{act}_{N\setminus C}^s \mid \exists a \in A \ \exists s' \in S \ \delta(\langle\!\langle a, b\rangle\!\rangle, s, s') > 0 \wedge s' \in x\Big\}
\end{aligned}
$$

where act_C^s is the set of available actions for coalition C at state s.

This can be computed via the following nested fixed point iteration:

$$
\begin{aligned}
y_0 &= v_{\text{U}} \\
y_n &= x_{n,\infty} && \forall n \geq 1 \\
x_{n,0} &= \emptyset && \forall n \geq 1 \\
x_{n,m} &= \big(\text{Apre}_C\big(y_{n-1}, x_{n,m-1}\big) \cap v(\varphi_1)\big) \cup v(\varphi_2) && \forall n, m \geq 1
\end{aligned}
$$

where v_{U} is the set of reachable states.

Upon convergence, we have $v(\langle\!\langle C\rangle\!\rangle P_{\max=1}(\varphi_1 \ \mathtt{U} \ \varphi_2)) = y_\infty$.

We can compute $\text{apre}_C(y, x)$ symbolically using BDD inputs $Y : S \to \mathbb{B}$ and $X : S \to \mathbb{B}$ as follows, yielding an operator of the form $\text{Apre}_C : (S \to \mathbb{B}) \times (S \to \mathbb{B}) \to (S \to \mathbb{B})$.

$$
\begin{aligned}
A_Y = \text{Avail}_C \wedge \text{ForAll}\Big(&\text{Act}_{N\setminus C}, \text{Implies}\Big(\text{Avail}_{N\setminus C}, \\
&\text{ForAll}(S', \\
&\text{Implies}(T_+, \text{ReplaceVars}(S \mapsto S', Y)))\Big)\Big) \\
B_X(A) = \text{Avail}_{N\setminus C} \wedge \text{ThereExists}(&\text{Act}_C, \text{Avail}_C \wedge A \wedge \\
&\text{ThereExists}(S', T_+ \wedge \text{ReplaceVars}(S \mapsto S', X))) \\
\text{Apre}_C(Y, X) = \text{ForAll}\Big(&\text{Act}_{N\setminus C}, \text{Implies}\Big(\text{Avail}_{N\setminus C}, B_X(A_Y)\Big)\Big)
\end{aligned}
$$

where:

$$\text{Avail}_D : S \times \text{Act}_D \to \mathbb{B} = \text{ThereExists}\big(\text{Act}_{N\setminus D}, \text{Avail}\big)$$
$$T_+ : S \times \text{Act}_C \times \text{Act}_{N\setminus C} \times S' \to \mathbb{B} = T > \text{Constant}(0)$$

Then, the rest of the fixed point iterations can be done as:

$$\begin{aligned} Y_0 &= V_{\text{U}} \\ Y_n &= X_{n,\infty} && \forall n \geq 1 \\ X_{n,0} &= \text{Constant}(0) && \forall n \geq 1 \\ X_{n,m} &= \big(\text{Apre}_C\big(Y_{n-1}, X_{n,m-1}\big) \wedge V(\varphi_1)\big) \vee V(\varphi_2) && \forall n, m \geq 1 \end{aligned}$$

Since both X and Y change monotonically, the nested fixed point iteration will converge in at most $|V_{\text{U}}|^2$ iterations.

5.1.3.2. Prob 0

The set of states that have maximum probability 0 of satisfying $\varphi_1 \ \texttt{U} \ \varphi_2$ can be computed as:

$$\begin{aligned} V(\langle\!\langle C \rangle\!\rangle P_{\max=0}(\varphi_1 \ \texttt{U} \ \varphi_2)) = V(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\texttt{G}\neg\varphi_2)) \vee \\ V(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\neg\varphi_2 \ \texttt{U} \ (\neg\varphi_1 \wedge \neg\varphi_2))) \end{aligned}$$

That is, these are the states where the players in $N \setminus C$ can always ensure either that φ_2 is never satisfied or that φ_2 remains unsatisfied until φ_1 is violated.

The second part $V(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\neg\varphi_2 \ \texttt{U} \ (\neg\varphi_1 \wedge \neg\varphi_2)))$ can be computed via the same pre-computation algorithm for probability 1 states described above, swapping the roles of the coalitions and replacing φ_1 and φ_2 with $\neg\varphi_2$ and $\neg\varphi_1 \wedge \neg\varphi_2$ respectively.

We compute $v(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\texttt{G}\neg\varphi_2))$ via the following (adapted from [24]):

$$v(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\texttt{G}\neg\varphi_2)) = \nu x.\big(\text{pre}_{N\setminus C}(x) \cap v(\neg\varphi_2)\big)$$

where:

$$\begin{aligned} \text{pre}_{N\setminus C}(X) = \Big\{ s \in S \mid \exists a_{N\setminus C} \in \text{act}^s_{N\setminus C} \forall a_C \in \text{act}^s_C \forall s' \in S \\ \big(\delta\big(\langle\!\langle a_C, a_{N\setminus C} \rangle\!\rangle, s, s'\big) > 0 \to s' \in X\big)\Big\} \end{aligned}$$

and ν is the greatest fixed point operator.

This can be computed via the following greatest fixed point iteration:

$$x_0 = v_{\mathrm{U}} \setminus v(\varphi_2)$$
$$x_n = \mathrm{pre}_{N \setminus C}(x_{n-1}) \cap (v_{\mathrm{U}} \setminus v(\varphi_2)) \ \forall n \geq 1$$

Then finally

$$v(\langle\!\langle C \rangle\!\rangle P_{\max=0}(\varphi_1 \ \mathtt{U} \ \varphi_2)) = x_\infty \cup v(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\neg\varphi_2 \ \mathtt{U} \ (\neg\varphi_1 \wedge \neg\varphi_2)))$$

The fixed point computation can be done symbolically via:

$$\begin{aligned} X_0 &= V_{\mathrm{U}} \wedge \neg V(\varphi_2) \\ X_n &= (V_{\mathrm{U}} \wedge \neg V(\varphi_2)) \wedge \mathrm{ThereExists}\Big(\mathrm{Act}_{N \setminus C}, \mathrm{Avail}_{N \setminus C} \wedge \\ &\qquad \mathrm{ForAll}\big(\mathrm{Act}_C, \mathrm{Implies}\big(\mathrm{Avail}_C, \\ &\qquad \mathrm{ForAll}\big(S', \\ &\qquad \mathrm{Implies}\big(T_+, \mathrm{ReplaceVars}(S \mapsto S', X_{n-1})\big)\big)\big)\big)\Big) \end{aligned}$$

Since X has states removed from it monotonically, this will finish in at most $|V_{\mathrm{U}}|$ iterations.

Finally, we combine both parts to get

$$V(\langle\!\langle C \rangle\!\rangle P_{\max=0}(\varphi_1 \ \mathtt{U} \ \varphi_2)) = X_\infty \vee V(\langle\!\langle N \setminus C \rangle\!\rangle P_{\max=1}(\neg\varphi_2 \ \mathtt{U} \ (\neg\varphi_1 \wedge \neg\varphi_2)))$$

Since this uses the Prob 1 algorithm, the total number of iterations taken is $O(|V_{\mathrm{U}}|^2)$.

5.1.3.3. Quantitative Value Iteration

After doing pre-computation, we will have three BDDs $V_{\max=0}, V_{\max=1}, V_? = V_{\mathrm{U}} \wedge \neg(V_{\max=0} \vee V_{\max=1})$ that correspond to states that have probability 0, probability 1, and unknown probability of satisfying the until property respectively.

We can then use the bounded-until value iteration algorithm to get $V_0, V_1, V_2, ...$ until we find that the values of $V_n : S \to [0, 1]$ converge.

While doing so, we restrict $T_?$ to matrix games for states in $V_?$ to reduce the computational load.

$$
\begin{aligned}
V_0 &= V_{\max=1} \\
V_n &= \text{IfThenElse}\Big(V_{\max=1}, \text{Constant}(1), \text{Val}\Big(Z_n, \text{Avail}_?, \text{Act}_C, \text{Act}_{N\setminus C}\Big)\Big) \\
\text{where } \text{Avail}_? &= \text{Avail} \wedge V_? \\
T_? &= \text{Restrict}(\text{Restrict}(T, \text{Avail}_?), \text{ReplaceVars}(S \mapsto S', V_{\text{U}})) \\
Z_n &= \text{MVMult}(T_?, \text{ReplaceVars}(S \mapsto S', V_{n-1}), S')
\end{aligned}
$$

Since $\text{Avail}_?$ does not include the states in $V_{\max=0}$, V_n is guaranteed to be 0 for states in $V_{\max=0}$, thus we do not need to explicitly set it to 0.

We test for convergence via

$$\text{supnorm}(V_n, V_{n-1}) < \varepsilon$$

where

- $\text{supnorm}(f, g) = \max_{s\in S}|f(s) - g(s)|$, i.e. the maximum absolute error
- $\varepsilon \in \mathbb{R}^+$ is a small constant representing the acceptable error

At this point, we can return $V(\langle\!\langle C\rangle\!\rangle P_{\max=?}(\varphi_1 \ \mathtt{U} \ \varphi_2)) = V_n$.

5.2. Maximum Expected Reward

As above, we deal with reward-based properties by focusing on computing $V(\langle\!\langle C\rangle\!\rangle R^r_{\max=?}(\rho))$, and do so by considering the structure of ρ.

Since each of the reward-based operators are checked only with reference to a single reward structure, we drop the r superscript for clarity.

We reuse the Val operator from Section 5.1 to solve the matrix games that arise.

5.2.1. Instantaneous Reward

To compute $V(\langle\!\langle C\rangle\!\rangle R_{\max=?}[\mathtt{I}^{=n}])$, we do

$$
\begin{aligned}
V_0 &= R_{\text{state}} \\
V_n &= \text{Val}\Big(Z_n, \text{Avail}, \text{Act}_C, \text{Act}_{N\setminus C}\Big) \\
\text{where } T_? &= \text{Restrict}(\text{Restrict}(T, \text{Avail}), \text{ReplaceVars}(S \mapsto S', V_{\text{U}})) \\
Z_n &= \text{MVMult}(T_?, \text{ReplaceVars}(S \mapsto S', V_{n-1}), S')
\end{aligned}
$$

and return $V(\langle\!\langle C\rangle\!\rangle R_{\max=?}[\mathtt{I}^{=n}]) = V_n$.

5.2.2. Bounded Cumulative Rewards

To compute $V(\langle\!\langle C\rangle\!\rangle R_{\max=?}[\mathtt{C}^{\leq n}])$, we do:

$$
\begin{aligned}
V_0 &= \text{Constant}(0) \\
V_n &= \text{Val}\big(Z_n, \text{Avail}, \text{Act}_C, \text{Act}_{N\setminus C}\big) + R_{\text{state}} \\
\text{where } T_? &= \text{Restrict}(\text{Restrict}(T, \text{Avail}), \text{ReplaceVars}(S \mapsto S', V_{\text{U}})) \\
R_{\text{trans?}} &= \text{Restrict}(R_{\text{trans}}, \text{Avail}) \\
Z_n &= R_{\text{trans?}} + \text{MVMult}(T_?, \text{ReplaceVars}(S \mapsto S', V_{n-1}), S')
\end{aligned}
$$

and return $V(\langle\!\langle C \rangle\!\rangle R_{\max=?}[\mathtt{C}^{\leq n}]) = V_n$.

5.2.3. Expected Reachability

To compute $V(\langle\!\langle C \rangle\!\rangle R_{\max=?}[\mathtt{F}\varphi])$, we first use $V_\infty = V(\langle\!\langle C \rangle\!\rangle P_{<1}(\mathtt{F}\varphi))$ via the pre-computation algorithm in Section 5.1.3.1.

We next construct $R_\gamma = (R^\gamma_{\text{state}}, R^\gamma_{\text{trans}})$ via

$$
\begin{aligned}
R^\gamma_{\text{state}} &= \text{IfThenElse}((R_{\text{state}} = 0) \wedge V_{\text{U}}, \text{Constant}(\gamma), R_{\text{state}}) \\
R^\gamma_{\text{trans}} &= \text{IfThenElse}(R_{\text{trans}} = 0, \text{Constant}(\gamma), R_{\text{trans}})
\end{aligned}
$$

Analogous to maximum probabilistic reachability, we focus only on states whose expected rewards are not known to be infinite or zero and use `Restrict` to minimise the size of the ADDs.

$$
\begin{aligned}
V_\varphi &= V(\varphi) \\
\text{Avail}_? &= \text{Avail} \wedge \neg V_\infty \wedge \neg V_\varphi \\
R^\gamma_{\text{state?}} &= R^\gamma_{\text{state}} \times \big(\neg V_\infty \wedge \neg V_\varphi\big) \\
R^\gamma_{\text{trans?}} &= \text{Restrict}\big(\text{Restrict}\big(R^\gamma_{\text{trans}}, \text{Avail}_?\big), \text{ReplaceVars}(S \mapsto S', V_{\text{U}})\big) \\
T_? &= \text{Restrict}(T, \text{Avail}_?)
\end{aligned}
$$

We then do the value iteration as follows:

$$
\begin{aligned}
V^\gamma_0 &= V_\infty \times \infty \\
V^\gamma_n &= \text{IfThenElse}\big(V_\infty, \infty, R^\gamma_{\text{state?}} + \text{Val}\big(Z_n, \text{Avail}_?, \text{Act}_C, \text{Act}_{N\setminus C}\big)\big) \\
\text{where } Z_n &= R^\gamma_{\text{trans?}} + \text{MVMult}\big(T_?, \text{ReplaceVars}\big(S \mapsto S', V^\gamma_{n-1}\big), S'\big)
\end{aligned}
$$

Since $\text{Avail}_?$ does not include the states in V_φ, V^γ_n is guaranteed to be 0 for states in V_φ and thus we do not need to explicitly set it to 0.

We repeat until convergence, tested via the `supnorm` operation as in Section 5.1.

We perform the second value iteration in a similar manner but with initial values determined by the final values from the previous value iteration, V_N^γ.

$$
\begin{aligned}
R_{\text{state?}} &= R_{\text{state}} \times \left(\neg V_\infty \wedge \neg V_\varphi\right) \\
R_{\text{trans?}} &= \text{Restrict}(\text{Restrict}(R_{\text{trans}}, \text{Avail}_?), \text{ReplaceVars}(S \mapsto S', V_{\text{U}})) \\
V_0 &= V_\infty \times \infty + V_N^\gamma \\
V_n &= \text{IfThenElse}\Big(V_\infty, \infty, R_{\text{state?}} + \text{Val}\Big(Z_n, \text{Avail}_?, \text{Act}_C, \text{Act}_{N \setminus C}\Big)\Big) \\
\text{where } Z_n &= R_{\text{trans?}} + \text{MVMult}(T_?, \text{ReplaceVars}(S \mapsto S', V_{n-1}), S')
\end{aligned}
$$

We use the same convergence criterion as before, returning $V(\langle\!\langle C \rangle\!\rangle R_{\max=?}[\mathtt{F}\varphi]) = V_N$ upon convergence.

Chapter 6

Symbolic Minimax via Symbolic Discounted Regret Matching

In this section, we describe a novel fully symbolic implementation of $\mathrm{Val}\big(Z, \mathrm{Avail}, \mathrm{Act}_C, \mathrm{Act}_{N\setminus C}\big)$ for the symbolic model checking algorithms in Chapter 5.

Rather than adapting from linear programming, we focus on regret matching, which has control flow independent of the payoff matrix entries, naturally allowing for a symbolic adaptation.

We first ignore coalitions and consider two-player zero-sum games with action sets A_A and A_B for players A (maximiser) and B (minimiser) respectively.

The payoff matrices for all states are represented as a single ADD

$$Z : S \to \mathrm{Act}_A \times \mathrm{Act}_B \to \mathbb{R}$$

Here, $Z(s) : \mathrm{Act}_A \times \mathrm{Act}_B \to \mathbb{R}$ represents the payoff matrix for the two-player zero-sum game at state s. Specifically, $Z(s)(a_A, a_B)$ gives the utility that player A receives when player A plays action a_A and player B plays action a_B at state s.

Our aim is to find a state-indexed ADD of minimax values: $\mathrm{Val}(Z, \mathrm{Avail}, \mathrm{Act}_A, \mathrm{Act}_B) : S \to \mathbb{R}$ where

- $\mathrm{Val}(Z, \mathrm{Avail}, \mathrm{Act}_A, \mathrm{Act}_B)(s)$ is the minimax value of solving the matrix game $Z(s)$ for each relevant state s. A relevant state is one with at least one valid action tuple, i.e. there exists $a_A \in \mathrm{Act}_A$ and $a_B \in \mathrm{Act}_B$ such that $\mathrm{Avail}(s, a_A, a_B) = 1$.
- $\mathrm{Val}(Z, \mathrm{Avail}, \mathrm{Act}_A, \mathrm{Act}_B)(s) = 0$ for irrelevant states.

We modify the discounted regret matching algorithm from Section 3.7 to solve all games represented in Z simultaneously using ADD operations. A crucial fact to handle is that not every action is available to each player at every state.

The goal is to generate (mixed) strategies for each player at each relevant state:

$$X_A : \mathrm{Act}_A \to S \to [0,1]$$
$$X_B : \mathrm{Act}_B \to S \to [0,1]$$

- We ensure that $X_p(a,s) = 0$ if action a is not available in state s for player p
- We ensure that $\sum_{a \in \mathrm{Act}_p} X_p(a,s) = 1$ for each relevant state

Initialisation:

We generate the BDD Avail_p that indicates which actions are available to player p at state s and use it to minimise the size of Z via `Restrict`.

$$\mathrm{Avail}_A : S \times \mathrm{Act}_A \to \mathbb{B} = \mathrm{ThereExists}(\mathrm{Act}_B, \mathrm{Avail})$$
$$\mathrm{Avail}_B : S \times \mathrm{Act}_B \to \mathbb{B} = \mathrm{ThereExists}(\mathrm{Act}_A, \mathrm{Avail})$$
$$Z_{\mathrm{Avail}} : S \to \mathrm{Act}_A \times \mathrm{Act}_B \to \mathbb{R} = \mathrm{Restrict}(Z, \mathrm{Avail})$$

We implement the uniform initialisation of current and zero initialisation of cumulative regrets and average strategies as follows:

$$X_p^{(0)} : \mathrm{Act}_p \to S \to [0,1] = \mathrm{Avail}_p \div \mathrm{Abstract}\big(+, \mathrm{Avail}_p, \mathrm{Act}_p\big)$$
$$R_p^{(0)} : \mathrm{Act}_p \to S \to \mathbb{R} = \mathrm{Constant}(0)$$
$$\sigma_p^{(0)} : \mathrm{Act}_p \to S \to \mathbb{R} = \mathrm{Constant}(0)$$

For each iteration $t \geq 1$:

1. We compute the utility for each player.

$$U_A^{(t)} : \mathrm{Act}_A \to S \to \mathbb{R} = \mathrm{MVMult}\Big(Z_{\mathrm{Avail}}, X_B^{(t-1)}, \mathrm{Act}_B\Big)$$
$$U_B^{(t)} : \mathrm{Act}_B \to S \to \mathbb{R} = -\ \mathrm{MVMult}\Big(Z_{\mathrm{Avail}}, X_A^{(t-1)}, \mathrm{Act}_A\Big)$$

2. We compute the payoff value for each player with the current strategies.

$$C_p^{(t)} : S \to \mathbb{R} = \mathrm{MVMult}\Big(X_p^{(t-1)}, U_p^{(t)}, \mathrm{Act}_p\Big)$$

3. We then compute the instantaneous regret.

$$I_p^{(t)} : \mathrm{Act}_p \to S \to \mathbb{R} = \Big(U_p^{(t)} - C_p^{(t)}\Big) \times \mathrm{Avail}_p$$

We mask this instantaneous regret with Avail_p to ensure that it is 0 for actions that are not available, including all actions in irrelevant states. This is important to ensure that unavailable actions do not contribute to the cumulative regret and thus the strategy updates.

4. We discount and accumulate the regret for each action for each player.

$$R_p^{(t)} : \text{Act}_p \to S \to \mathbb{R} = \text{IfThenElse}\left(R_p^{(t-1)} \geq 0, \frac{t^\alpha}{1+t^\alpha} R_p^{(t-1)}, \frac{t^\beta}{1+t^\beta} R_p^{(t-1)}\right) + I_p^{(t)}$$

5. We compute the new strategy for each player by normalising the positive cumulative regret.

$$\begin{aligned} Y_p^{(t)} : \text{Act}_p \to S \to \mathbb{R}_0^+ &= \text{Max}\Big(R_p^{(t)}, 0\Big) \\ X_p^{(t)} : \text{Act}_p \to S \to [0,1] &= \text{IfThenElse}\Big(\text{Abstract}\Big(+, Y_p^{(t)}, \text{Act}_p\Big) > 0, \\ &\qquad Y_p^{(t)} \div \text{Abstract}\Big(+, Y_p^{(t)}, \text{Act}_p\Big), X_p^{(0)}\Big) \end{aligned}$$

6. We discount and update the accumulated strategies for each player.

$$\sigma_p^{(t)} : \text{Act}_p \to S \to \mathbb{R}_0^+ = \left(\frac{t}{t+1}\right)^\gamma \sigma_p^{(t-1)} + X_p^{(t)}$$

7. We compute the average strategy for each player.

$$\overline{X}_p^{(t)} : \text{Act}_p \to S \to [0,1] = \sigma_p^{(t)} \div \text{Abstract}\Big(+, \sigma_p^{(t)}, \text{Act}_p\Big)$$

8. We can check for convergence every $k = 1000$ iterations via:

$$\begin{aligned} \text{UpperBound}^{(t)} : S \to \mathbb{R} = \\ \text{Abstract}\Big(\max, \text{MVMult}\Big(Z_{\text{Avail}}, \overline{X}_B^{(t)}, \text{Act}_B\Big) - \infty \times (1 - \text{Avail}_A), \text{Act}_A\Big) \\ \text{LowerBound}^{(t)} : S \to \mathbb{R} = \\ \text{Abstract}\Big(\min, \text{MVMult}\Big(\overline{X}_A^{(t)}, Z_{\text{Avail}}, \text{Act}_A\Big) + \infty \times (1 - \text{Avail}_B), \text{Act}_B\Big) \end{aligned}$$

From here, we can compute the difference as $\text{UpperBound} - \text{LowerBound}$. We can then determine the states that have converged, which can be represented by the ADD

$$\text{Finished}^{(t)} : S \to \mathbb{B} = \text{UpperBound} - \text{LowerBound} < \varepsilon$$

We can then accumulate the finished results in an ADD

$$\begin{aligned}
\text{Result}^{(t)} : S \to \mathbb{R} &= \text{MVMult}\Big(\overline{X}_A^{(t)}, \text{MVMult}\Big(Z_{\text{Avail}}, \overline{X}_B^{(t)}, \text{Act}_B\Big), \text{Act}_A\Big) \\
\text{Val}(Z)^{(0)} : S \to \mathbb{R} &= \text{Constant}(0) \\
\text{Val}(Z)^{(t)} : S \to \mathbb{R} &= \text{Val}(Z)^{(t-1)} + \text{Result}^{(t)} \times \text{Finished}^{(t)}
\end{aligned}$$

A subtle point is that the computation of $\text{Result}^{(t)}$ zeros out the result for irrelevant states since Z_{Avail} is multiplied with $\overline{X}_B^{(t)}$ within $\text{MVMult}\Big(Z_{\text{Avail}}, \overline{X}_B^{(t)}, \text{Act}_B\Big)$ and $\overline{X}_B^{(t)}(s, \vec{a}_B)$ is always 0 for irrelevant states s. Thus, even though `Finished` will contain irrelevant states, this does not affect $\text{Val}(Z)^{(t)}$.

We also remove these states from all the relevant BDDs/ADDs to ensure that we do not add them back in future iterations and to reduce the computational load for future iterations.

$$\begin{aligned}
X_p^{(0)} &= \neg\text{Finished}^{(t)} \times X_p^{(0)} \\
X_p^{(t)} &= \neg\text{Finished}^{(t)} \times X_p^{(t)} \\
R_p^{(t)} &= \neg\text{Finished}^{(t)} \times R_p^{(t)} \\
\sigma_p^{(t)} &= \neg\text{Finished}^{(t)} \times \sigma_p^{(t)} \\
\text{Avail}_p &= \neg\text{Finished}^{(t)} \times \text{Avail}_p \\
Z_{\text{Avail}} &= \text{Restrict}(\text{Restrict}(Z_{\text{Avail}}, \text{Avail}_A), \text{Avail}_B)
\end{aligned}$$

9. The final minimax values are $\text{Val}(Z)^{(T)}$, where T is the final iteration.

6.1. Dealing with Coalitions

To deal with two non-empty coalitions of players C and $N \setminus C$, we observe that the algorithm does not depend on how the individual actions of each player are encoded as ADD variables, only on there being two disjoint action sets: one for the maximising player and one for the minimising player.

Thus, we can take $\text{Act}_A = \text{Act}_C$ and $\text{Act}_B = \text{Act}_{N \setminus C}$, i.e.

$$\prod_{i \in N} \text{Act}_i = \left(\prod_{i \in C} \text{Act}_i\right) \times \left(\prod_{i \in N \setminus C} \text{Act}_i\right) = \text{Act}_C \times \text{Act}_{N \setminus C}$$

This means that we can use Z directly without modification:

$$Z : S \times \prod_{i \in N} \text{Act}_i \to \mathbb{R} = S \times \text{Act}_C \times \text{Act}_{N \setminus C} \to \mathbb{R}$$

Furthermore, we can view T as a transition relation for two players as well:

$$T : \prod_{i \in N} \text{Act}_i \times S \times S' \to [0, 1] = \text{Act}_C \times \text{Act}_{N \setminus C} \times S \times S' \to [0, 1]$$

Thus, all of the above regarding two-player games can be generalised to two-coalition games with no modification.

However, this method does not correctly handle total ($C = N$) or empty ($C = \emptyset$) coalitions. Fortunately, such coalitions reduce the minimax problem to a simple max or min problem, which can be solved directly. We still need to ensure that we only consider available actions and return zero for irrelevant states.

Algorithm 1: Total/Empty Coalition Val

```
   procedure valTrivial(
1                Z: ADDNode, avail: ADDNode,
                 maxiVars: List[ADDVar], miniVars: List[ADDVar])
2    if miniVars.isEmpty() then
3      ZMasked ← IfThenElse(avail, Z, −∞)
4      result ← Abstract(Max, ZMasked, maxiVars)
5      relevantStates ← ThereExists(maxiVars, avail)
6      return result × relevantStates
7    else if maxiVars.isEmpty() then
8      ZMasked ← IfThenElse(avail, Z, ∞)
9      result ← Abstract(Min, ZMasked, miniVars)
10     relevantStates ← ThereExists(miniVars, avail)
11     return result × relevantStates
12   else
13     assert false
14   end if
15 end procedure
```

This is implemented in Algorithm 1 where we use $\pm\infty$ to avoid impacting the min/max abstract. We zero out these $\pm\infty$ values to ensure they do not propagate into the result via the multiplication by `relevantStates`.

6.2. Dynamic Variable Reordering

A benefit of this fully symbolic implementation is that it is independent of variable reorderings. Thus, we can safely enable automatic variable reordering throughout the checking of each path/reward formula. Based on ad hoc testing, group sifting [43] provided a good balance between speed and effectiveness.

Chapter 7

Hybrid Minimax via Linear Programming

In this section, we describe an efficient hybrid implementation of $\mathrm{Val}\big(Z, \mathrm{Avail}, \mathrm{Act}_C, \mathrm{Act}_{N\setminus C}\big)$ for the model checking algorithms in Chapter 5.

This algorithm relies on action ADD variables being ordered after current-state variables. The core idea is that sub-ADDs rooted at action variables represent single matrix games, and the relevant roots are those that are also cofactors of current-state nodes (the coloured nodes in Figure 1). For each such sub-ADD, we extract the matrix game explicitly, solve it via linear programming, and substitute the subtree with a constant representing the matrix-game value (the square coloured nodes in Figure 1).

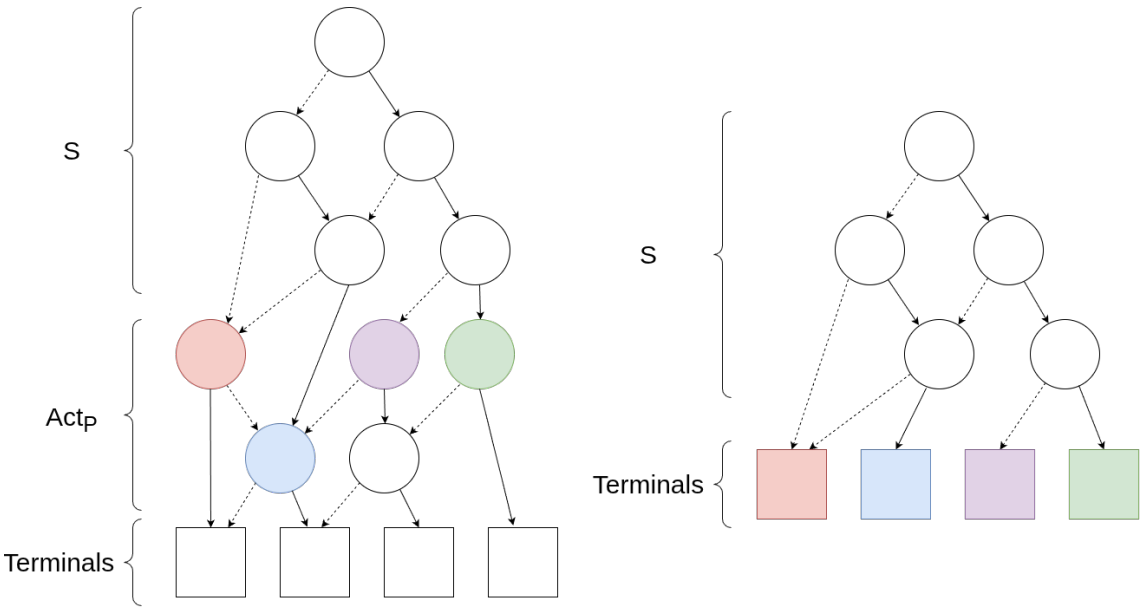

Figure 1: Illustration of Relevant Sub-ADDs and Substitution

As a pre-processing step, we integrate the action availability information from Avail into Z by replacing invalid action tuples with the sentinel value $-\infty$.

$$Z = \mathrm{IfThenElse}(\mathrm{Avail}, Z, -\infty)$$

The first step is finding the roots of the relevant matrix sub-ADDs. We do so with Algorithm 2 by recursing over Z from the root. We can identify such roots as a node whose variable is the first action variable encountered in a path from the root to that node. We memoise visited nodes to avoid repeated traversal of the same sub-ADD.

Algorithm 2: Find Matrix Games

```
   procedure FindMatrixGames(
1                  u: ADDNode, roots: Set[ADDNode],
                   visited: Set[ADDNode], actV: Set[ADDVar])
2  | if u is terminal or u ∈ visited then
3  |  | return
4  | end if
5  | visited ← visited ∪ {u}
6  | if u.var ∈ actV then
7  |  | roots ← roots ∪ {u}
8  | else
9  |  | FindMatrixGames(u.high, roots, visited, actV)
10 |  | FindMatrixGames(u.low, roots, visited, actV)
11 | end if
12 end procedure
```

For each identified matrix-game root node, we use Algorithm 3 to build an explicit matrix game by considering all paths along the matrix-game sub-ADD. For each path, we compute the components of the action tuple belonging to the minimiser and maximiser, as well as the payoff from the action tuple. This is used to fill in a payoff matrix.

Algorithm 3: Build Matrix Game

```
   procedure BuildMatrixGame(
1                  u: ADDNode, maxiVars: List[ADDVar],
                   miniVars: List[ADDVar])
2  | entryMaxi ← List[]
3  | entryMini ← List[]
4  | entryPayoff ← List[]
5  | for each non −∞ (expanded) path p ∈ u:
6  |  | entryMaxi.add(p.restrictTo(maxiVars))
7  |  | entryMini.add(p.restrictTo(miniVars))
8  |  | entryPayoff.add(p.terminalValue())
9  | end for
```

```
    allMaxActions ← entryMaxi.sorted().unique()
    allMinActions ← entryMini.sorted().unique()
    matrixGame ← Matrix.OfSize(
                    allMaxActions.size(), allMinActions.size())
    for each (eMaxi, eMini, payoff) ∈ zip(entryMaxi, entryMini, entryPayoff):
        i ← allMaxActions.indexOf(eMaxi)
        j ← allMinActions.indexOf(eMini)
        matrixGame[i][j] ← payoff
    end for
    return matrixGame
end procedure
```

For each matrix game, we use the same linear-programming formulation as the explicit-state implementation [30] to find its minimax value. Finally, we use Algorithm 4 to generate a new ADD that represents the substitution of matrix subtrees with their minimax values. We ensure that any $-\infty$ sentinel values are replaced with 0.

Algorithm 4: Substitute ADD Subtrees with Minimax Values

```
procedure substituteTrees(
                u: ADDNode, actTrees: Map[ADDNode, Double],
                memo: Map[ADDNode, ADDNode])
    if u ∈ actTrees then
        return Constant(actTrees[u])
    else if u ∈ memo then
        return memo[u]
    else if u is terminal then
        return u = Constant(−∞) ? Constant(0) : u
    else
        newHigh ← substituteTrees(u.high, actTrees, memo)
        newLow ← substituteTrees(u.low, actTrees, memo)
        newNode ← IfThenElse(u.var, newHigh, newLow)
        memo[u] ← newNode
        return newNode
    end if
end procedure
```

These steps are combined to form `HybridVal` in Algorithm 5, taking into account the edge case of a total/empty coalition which uses Algorithm 1 instead.

Algorithm 5: Hybrid Val Algorithm

```
   procedure hybridVal(
1                  Z: ADDNode, avail: ADDNode,
                   maxiVars: List[ADDVar], miniVars: List[ADDVar])
2    if maxiVars.isEmpty() || miniVars.isEmpty() then
3     return valTrivial(Z, avail, maxiVars, miniVars)
4    end if
5    Z ← IfThenElse(avail, Z, -infinity)
6    roots ← Set[]
7    visited ← Set[]
8    FindMatrixGames(Z, roots, visited, maxiVars ∪ miniVars)
9    actTrees ← Map[]
10   for each root r ∈ roots:
11     matrixGame ← BuildMatrixGame(r, maxiVars, miniVars)
12     value ← SolveMinimaxLP(matrixGame)
13     actTrees[r] ← value
14   end for
15   memo ← Map[]
16   return substituteTrees(Z, actTrees, memo)
17 end procedure
```

As shown later in Section 8.3, the translation between symbolic and explicit representations is usually not a dominant cost relative to other hotspots of the code, making it our preferred method for implementing `Val`.

7.1. Dynamic Variable Reordering

With this approach, we cannot arbitrarily reorder the ADD variables. However, we can still form two groups: $S \cup S'$ and $\bigcup_p \text{Act}_p$, then permit reordering amongst the variables within each group, keeping the first group above the second group.

The implementation of hybrid `Val` requires the variable ordering to remain fixed while it runs, so we trigger dynamic reordering only once per path/reward formula checked. As before, we use group sifting [43].

Chapter 8

Experimental Results and Analysis

We evaluate the performance of different variants of our symbolic verification algorithms on a set of real-world case studies from the PRISM-games repository, described in Table 2.

All reported timings are wall-clock times measured within PRISM-games, split across model construction, pre-computation, and value iteration. Memory usage is reported as the maximum resident set size measured using the Unix `time` utility, capturing both Java heap usage and native memory consumption by the CUDD library.

Each experiment is reported as the average of three runs using the default PRISM-games configuration (1 GB Java heap, 1 GB CUDD memory) with OpenJDK 25.0.2+10 on a machine running NixOS 26.05 with an Intel i7-11800H CPU.

Name	**Abbrev**	**Params**	**Description**
Jamming 4/6	Jam4/6	`slots`	A network with 4 or 6 channels where a jammer interferes with a user's transmissions [6, 22].
Aloha Backoff 3	Aloha	`D, bcmax, q`	A wireless network with 3 users sending packets via the slotted ALOHA protocol [8].
Power Control 2	Pow	`emax, fail, powmax`	A model of phones controlling their signal power in a cellular network [8, 44].
IDS Simple	IDS	`K, rounds`	A model of the interaction between an intrusion detection policy and an attacker [22, 45].

Name	Abbrev	Params	Description
Robot Coordination 2	Robot	`l, q`	A model of two robots concurrently moving in a grid [8].
Public Good	PubGood	`einit, emax, f, kmax`	A repeated public good game, a well-studied model in economics [5].
User Centric	UserCent	`K, td`	A model of users cooperating in a network [22].

Table 2: CSG Case Studies

8.1. Experimental Results for Model Construction

We first compare the performance of symbolic and explicit-state model construction, isolating the cost of constructing the transition system and reward structures.

Model construction involves parsing the input file, building the transition system and reward structures, and then performing reachability to filter out irrelevant states. The explicit-state implementation is the one described in [30] and the symbolic implementation is the one described in Chapter 4.

For larger instances, either implementation may run out of memory (denoted by `memout`).

Case Study	Args.	States	Symbolic			Explicit	
			ADD Nodes	Cons. Time (s)	Mem (MB)	Cons. Time (s)	Mem (MB)
Jam4	25	5965	8559	**0.02**	**131.7**	1.52	500.3
	50	21915	17031	**0.03**	**130.2**	5.03	784.0
	75	47865	28070	**0.03**	**130.3**	11.89	1249.4
Aloha	5,3,0.8	51200	27970	**0.10**	**145.2**	0.59	376.2
	5,4,0.8	227243	41968	**0.13**	**144.5**	2.04	641.8
	5,6,0.8	3725898	63500	**0.24**	**158.1**	memout	memout

Case Study	Args.	States	Symbolic			Explicit	
			ADD Nodes	Cons. Time (s)	Mem (MB)	Cons. Time (s)	Mem (MB)
Pow	50,0.1,50	69386	12642	**0.10**	**133.7**	0.88	535.8
	75,0.1,75	245979	25417	**0.21**	**138.2**	2.53	1037.3
	100,0.1,100	604880	39361	**0.46**	**145.7**	memout	memout
IDS	10000,10000	20001	137	1.90	**137.1**	**0.40**	312.1
	20000,20000	40001	146	6.97	**146.6**	**0.60**	445.9
	40000,40000	80001	155	25.99	**167.3**	**0.95**	517.0
Robot	10,0.1	9802	2149	**0.02**	**129.3**	0.51	339.1
	20,0.1	159202	2728	**0.02**	**130.5**	memout	memout
Pub-Good	5,5,1,5	1005	13297	**0.03**	**131.5**	0.30	249.0
	10,10,1,10	11457	106657	**0.19**	**145.1**	1.53	465.9
	20,20,1,20	112538	759373	**3.63**	**267.6**	memout	memout
User-Cent	1,1	1409	NA	memout	memout	**0.11**	**210.9**
	2,2	7825	NA	memout	memout	**0.29**	**260.5**

Table 3: Model Construction Results

Table 3 shows that symbolic model construction is almost always faster than the explicit-state approach and, in almost all cases, significantly more memory efficient. This behaviour is consistent with prior results for symbolic model checking of MDPs [13] and TSGs [28]. The main exception is the User-Centric case study, where the symbolic implementation runs out of memory on the tested instances. We believe this model is ill-suited to ADD representation because it produces a wide variety of probability values as complex functions of the state, resulting in sparse ADDs with little sharing.

The primary reason for the improved speed is that symbolic construction builds the transition system in a compositional manner using ADD operations, avoiding explicit enumeration of the state space. In addition, symbolic reachability extends the set of

reachable states simultaneously across all states, whereas the explicit-state approach must iterate over states individually. The memory efficiency can be explained by the more compact representation of the transition system and sets used during reachability.

8.2. Experimental Results for Model Checking

We now compare the performance of different complete implementations of the model checking algorithms, listed in Table 4. We report memory usage and the timing statistics described in Table 5.

Type	Variant	Description
LP-based	Explicit	The existing explicit-state implementation, described in [30].
	Explicit with Cache	The explicit-state implementation augmented with a hash table that stores the minimax values of previously solved matrix games, avoiding repeated solves of the same game.
	Symbolic	The symbolic model checking algorithm as described in Chapter 5, which uses the hybrid minimax algorithm in Chapter 7 to solve the matrix games.
Regret-based	Explicit with Regret Matching	The same explicit-state representation as the explicit-state implementation but with the discounted regret matching algorithm in Section 3.7 to solve the matrix games.
	Symbolic with Regret Matching	The same symbolic representation as the symbolic implementation but with the symbolic regret matching algorithm in Chapter 6 to solve the matrix games.

Table 4: CSG Model Checking Implementation Variants

Timing	Description
Qualitative Time	The time taken for qualitative checks, i.e. the pre-computation time for infinite horizon properties

Timing	Description
Quantitative Time	The time taken for quantitative checks, i.e. the value iteration time
Model Checking Time	The total time taken for model checking, i.e. Qualitative Time + Quantitative Time
Construction Time	The time taken for model construction, as reported in Table 3
Total Time	The total time taken for model construction and model checking, i.e. Construction Time + Model Checking Time

Table 5: Model Checking Time Statistics

The model checking algorithms are run on smaller test cases than in Table 3 since the model checking process is more computationally intensive.

Preliminary testing showed that regret matching was much slower, so we benchmark the LP and regret-matching variants at different scales.

For each case study and property, we try to select parameters such that the longest runtime among the compared variants is at least 5 seconds. The exception is the Jamming case study: beyond 23 slots, the linear programming solver fails with a numerical error when solving the matrix games.

We first consider the linear programming variants.

Case Study Prop ID Prop Type	Args	Symbolic		Explicit		Explicit w Caching	
		Total Time (s)	Mem (MB)	Total Time (s)	Mem (MB)	Total Time (s)	Mem (MB)
Jam4 Pr 1 `Pmax=? [F..]`	10	**0.07**	**147.9**	0.95	347.7	0.79	354.7
	23	**0.34**	**178.4**	5.43	553.7	5.05	556.7
Jam6 Pr 1 `Pmax=? [F..]`	10	**0.08**	**145.6**	3.07	566.4	3.02	569.4
	23	**0.43**	**176.7**	25.86	1286.8	24.49	1266.8

Case Study Prop ID Prop Type	Args	Symbolic		Explicit		Explicit w Caching	
		Total Time (s)	Mem (MB)	Total Time (s)	Mem (MB)	Total Time (s)	Mem (MB)
Jam6 Pr 2 `Rmax=? [F..]`	10	**0.38**	**182.7**	3.73	581.6	3.30	590.6
	23	**5.56**	**369.9**	23.59	1282.5	19.30	1283.3
Aloha Pr 3 `Pmax=? [F..]`	4,4,0.8	**0.78**	**186.5**	4.49	688.8	4.49	730.3
	5,4,0.8	**0.93**	**192.9**	7.10	882.0	6.95	844.2
Aloha Pr 1 `Rmin=? [F..]`	2,2,0.8	1.34	**254.7**	0.98	345.4	**0.94**	348.6
	3,3,0.8	9.49	**352.1**	**3.00**	406.2	6.32	591.1
Robot Pr 1 `Pmax=? [..U..]`	10,0.1	**2.29**	**312.9**	4.83	545.5	4.61	565.3
	13,0.1	**9.23**	**599.0**	22.04	764.8	20.80	737.2
Robot Pr 3 `Rmin=? [F..]`	8,0.1	2.83	**302.9**	**1.95**	368.1	1.97	384.8
	10,0.1	10.01	**517.4**	4.86	573.9	**4.43**	548.0
IDS Pr 1 `Rmin=? [F..]`	250,250	2.11	**269.9**	3.37	323.7	**1.57**	326.0
	500,500	8.17	**279.4**	12.63	349.9	**5.15**	354.0
IDS Pr 2 `Rmin=? [F..]`	250,250	**0.71**	**220.7**	2.96	324.4	1.01	328.8
	500,500	**2.48**	**270.8**	10.98	347.7	2.67	350.9
	750,750	5.44	**270.1**	26.91	383.7	**5.12**	337.6

Table 6: Comparison of Total Time and Memory Usage for LP-Based Model Checking Algorithms

From Table 6, the symbolic algorithm uses less memory across all benchmarks. For all benchmarks that involve calculating a maximum/minimum probability, the symbolic algorithm is also faster. The memory improvement can be attributed to the more compact symbolic representation of sets and state vectors. The symbolic algorithm is also favoured for probabilistic properties. We hypothesise that this is because probabilistic properties

tend to yield fewer distinct terminals, allowing ADDs to be more compact and efficient than in reward properties.

To better understand the source of the timing improvements, we broke down the total time into phases in Table 9 and Table 10 in the Appendix. Table 9 shows consistency with the model construction results: the symbolic implementation performs better in the model construction phase. From Table 10, we see that the qualitative part runs much faster in the symbolic algorithm than the explicit-state algorithm. This accurately reflects the success of symbolic model checking in non-probabilistic settings, where qualitative algorithms are the main focus.

Thus, the symbolic algorithm is typically limited by its performance in the quantitative phase. Section 8.3 investigates this bottleneck. However, there are some cases where the quantitative part of the symbolic algorithm performs better than the explicit-state implementation despite needing to convert between symbolic and explicit representations.

There are two main sources of these time savings. First, in the symbolic implementation, the entries of matrix Z are computed for all states at once using `MVMult`, whereas the explicit implementations compute them state by state. Second, repeated matrix games are represented by the same ADD subtree, so their minimax values do not need to be recomputed. Comparison with the explicit-state cached implementation suggests that both effects contribute to the symbolic speed-ups: in some cases the symbolic implementation outperforms both explicit-state variants in the quantitative phase, while in others it outperforms only the uncached explicit-state implementation.

Overall, although explicit-state implementations are faster in some cases, the symbolic implementation often reduces memory usage and can improve runtime, especially on structure-heavy models with many repeated matrix games.

We now look at the regret matching variants.

Case Study Prop ID Prop Type	**Args**	**Symbolic w Reg Matching**				**Explicit w Reg Matching**			
		Qual Time (s)	**Quant Time (s)**	**Total Time (s)**	**Mem (MB)**	**Qual Time (s)**	**Quant Time (s)**	**Total Time (s)**	**Mem (MB)**
	10	**0.00**	29.38	29.39	**297.0**	0.33	**17.07**	**17.85**	463.0

Case Study Prop ID Prop Type	Args	Symbolic w Reg Matching				Explicit w Reg Matching			
		Qual Time (s)	Quant Time (s)	Total Time (s)	Mem (MB)	Qual Time (s)	Quant Time (s)	Total Time (s)	Mem (MB)
Jam4 Pr 1 `Pmax=? [F..]`	20	**0.01**	109.49	109.52	**318.4**	2.06	**74.47**	**77.60**	507.3
Jam6 Pr 1 `Pmax=? [F..]`	10	**0.00**	**36.63**	**36.65**	**303.0**	0.98	39.92	42.74	554.2
	20	**0.01**	**138.96**	**138.99**	**316.7**	9.21	193.83	209.70	1063.5
Jam6 Pr 2 `Rmax=? [F..]`	3	**0.00**	**0.70**	**0.71**	**135.9**	0.05	6.37	6.79	483.5
	4	**0.00**	**1.33**	**1.34**	**141.8**	0.08	15.66	16.25	468.8
Aloha Pr 3 `Pmax=? [F..]`	2,2,0.8	0.24	0.83	1.13	**146.8**	**0.21**	**0.12**	**0.47**	340.1
	3,2,0.8	**0.21**	55.15	55.42	**310.6**	0.28	**0.45**	**0.93**	432.4
Robot Pr 1 `Pmax=? [..U..]`	3,0.1	**0.00**	2.40	2.42	**162.9**	0.02	**0.51**	**0.56**	361.8
	4,0.1	**0.00**	9.66	9.68	**174.5**	0.14	**3.22**	**3.50**	389.1
Robot Pr 3 `Rmin=? [F..]`	2,0.1	**0.00**	**0.00**	**0.02**	**129.9**	0.01	0.06	0.09	169.1
	3,0.1	**0.00**	27.64	27.66	**289.7**	0.01	**0.61**	**0.66**	357.7
IDS Pr 1 `Rmin=? [F..]`	3,3	**0.00**	34.87	34.88	**298.0**	0.01	**1.92**	**1.94**	359.6
	4,4	**0.00**	53.48	53.50	**292.4**	0.01	**3.35**	**3.38**	337.9
IDS Pr 2 `Rmin=? [F..]`	3,3	**0.00**	14.27	14.28	**284.9**	0.01	**0.70**	**0.73**	345.8
	4,4	**0.00**	18.94	18.96	**286.6**	0.01	**0.82**	**0.84**	337.4

Table 7: Comparison of Regret-Based Model Checking Algorithms

We do not focus on model construction time here since model construction time is negligible for these smaller benchmarks.

Similar to the LP-based variants, symbolic regret matching uses significantly less memory than its explicit-state counterpart. However, its execution time is substantially slower in most cases. This is likely because iterative regret updates produce many distinct payoff values across states, which prevents effective ADD sharing and results in large, sparse

decision diagrams that are costly to manipulate. In contrast, the explicit-state approach evaluates only one matrix at a time, which is much more efficient.

Although a purely symbolic approach via regret matching is conceptually appealing, it is currently not competitive for model checking CSGs.

8.3. Profiling

Profiling of the hybrid symbolic model checking algorithm across multiple case studies reveals that when the symbolic implementation runs significantly slower, ADD operations appear to be the main bottleneck.

We use `async-profiler` [46], a sampling profiler that tracks both Java and native stack frames, allowing us to capture time spent in both Java and native CUDD code. We analyse its flame graph output, which visualises call stacks and time spent in each method. We only track CPU time rather than wall time to prevent interference by other processes running on the machine.

While this section is presented after the methodology and experimental results, profiling was used to guide optimisation of the implementation. For example, it identified the need to reserve sufficient space for the hashmaps used in Algorithm 5 to avoid costly resizing operations, which was a major bottleneck in early versions of the implementation. The flame graphs presented here reflect the final implementation.

Figure 2 and Figure 3 are flame graphs from model checking the largest parameters of the two least favourable benchmarked properties from Table 6. These are the first property of 'Aloha Backoff 3' (`<<usr1>>R{"time"}min=?[F s1=3]`) and the third property of 'Robot Coordination 2' (`<<robot1>>R{"time1"}min=? [F "goal1"]`).

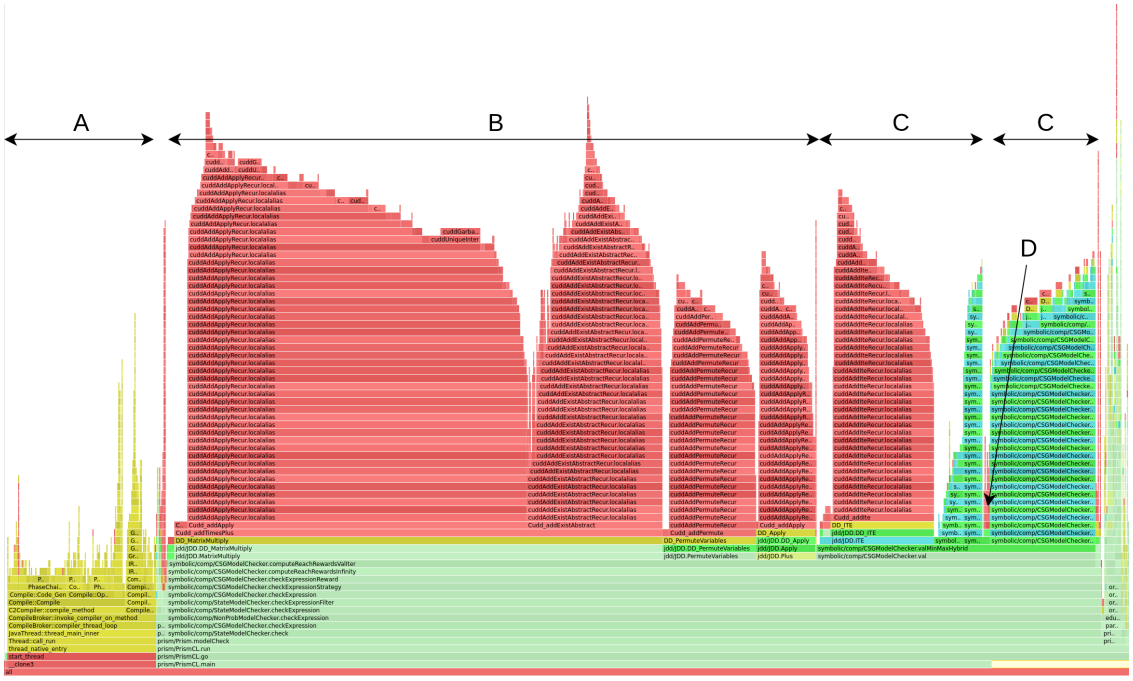

Figure 2: Flame Graph for Symbolic Model Checking of Aloha Backoff 3 Property 1

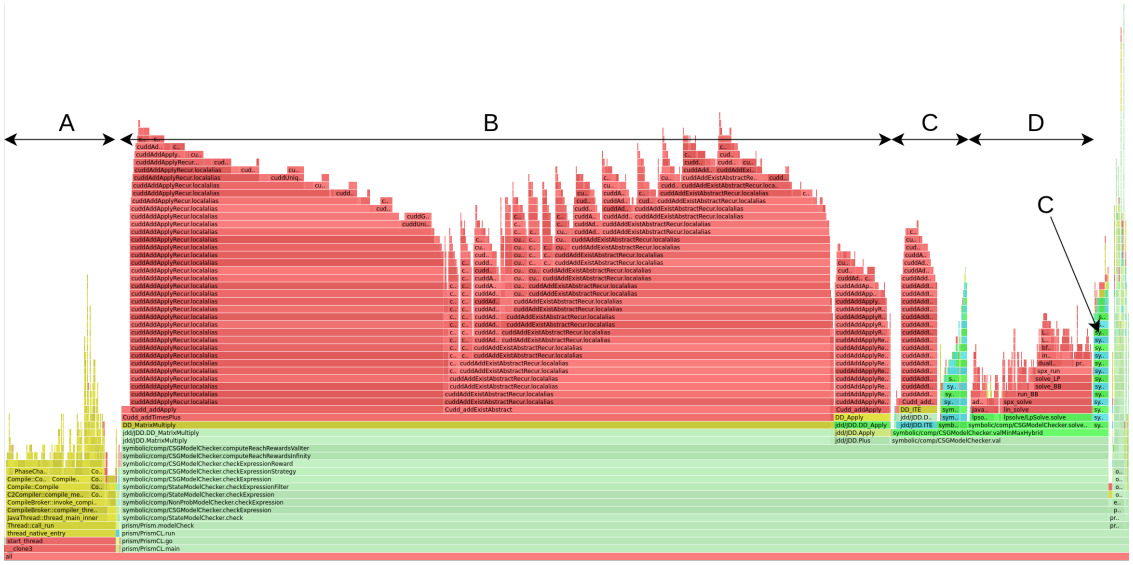

Figure 3: Flame Graph for Symbolic Model Checking of Robot Coordination 2 Property 3

Each flame graph can be divided roughly into five main regions.

Section	Description
A	Time spent outside the main thread, particularly on garbage collection.

Section	Description
B	Time spent doing symbolic operations like `MVMult` and `ReplaceVars`, which are used to construct `Z` symbolically for all states.
C	Time within hybrid `Val` not doing LP solving: pre-processing, building each distinct matrix game represented by a relevant sub-ADD of `Z` and rebuilding the ADD after solving the matrix game.
D	Time spent building the linear program for each matrix game and solving it with the LP solver.
Unlabelled	The rest of the time, including model construction

Table 8: Sections of the Flame Graph for Symbolic Model Checking Executions

We can observe that in these cases, the ADD operations in region B, together with the `IfThenElse` pre-processing step for `Val`, account for most of the runtime.

These results indicate that the translation between the symbolic representation and the explicit matrix games in Algorithm 5 is not a major bottleneck for these test cases, as the time taken to build the matrix games and to rebuild the ADDs after solving the matrix games is relatively small in both cases.

Improving performance further will likely require reducing the cost of ADD operations, for example through better variable orderings or a `Val` procedure that does not require action variables to appear at the bottom of the ADD.

Chapter 9

Conclusion

In this thesis, we have developed algorithms and an implementation of a symbolic model checking approach for CSGs. To the best of our knowledge, this is the first symbolic model checking algorithm for CSGs.

We have shown that the proposed hybrid model checking algorithm can outperform the existing explicit-state approach implemented in PRISM-games in terms of both memory usage and, in many cases, runtime across a range of representative case studies. These results demonstrate that symbolic techniques, which have been highly successful for other probabilistic models, can also be effectively applied to two-coalition zero-sum CSG model checking.

In addition, we have presented and evaluated a fully symbolic model checking algorithm based on symbolic discounted regret matching. While this approach is algorithmically well-founded and conceptually appealing, our experimental results indicate that it is currently less practical than the hybrid alternative. Nevertheless, it provides an important step towards fully symbolic minimax computation for CSGs and highlights several promising directions for further optimisation.

Finally, through detailed profiling of our implementation, we have identified that in poorly performing benchmarks, the primary performance bottleneck is ADD operations rather than the translation between symbolic and explicit representations. This insight provides clear guidance for future work aimed at improving the scalability and performance of symbolic verification for CSGs.

9.1. Further Work

First, we have restricted our focus to a subset of the full rPATL logic for CSGs. In particular, we have not considered properties involving more than two coalitions, general-sum objectives or multi-objective properties. While existing explicit-state techniques for

some of these extensions have been explored in the literature [11], it remains an open question how effectively they can be adapted to a symbolic setting.

Next, our implementation explicitly avoids multi-threading to provide a fair comparison with the single-threaded implementation of the explicit-state approach. However, there are many opportunities to parallelise both algorithmic approaches. It is particularly interesting to consider multi-threaded decision diagram libraries like Sylvan [47] and OxiDD [48]. Future work could therefore optimise both approaches for multi-threading and compare their performance.

Another important direction for future work is strategy synthesis. Although the algorithms developed in this thesis compute information from which optimal or near-optimal strategies may be derived, additional work is required to extract and present these strategies in a form suitable for practical use.

There is also considerable scope for improving the regret-matching-based symbolic approach. Regret minimisation continues to be an active area of research, with several recent algorithmic variants proposed in the literature [49]. Investigating symbolic versions of these approaches, and comparing them empirically with discounted regret matching, is a promising avenue for future research.

Finally, symbolic model checking could be extended to richer variants of CSGs, such as robust CSGs [50], which explicitly model uncertainty in transition probabilities. Extending symbolic techniques to these models would further broaden the applicability of formal verification for complex multi-agent systems.

Bibliography

1. Parker, D.: Multi-Agent Verification and Control with Probabilistic Model Checking. In: Proc. 20th International Conference on Quantitative Evaluation of SysTems (QEST'23). pp. 1–9. Springer (2023).

2. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic Model Checking: Applications and~Trends. In: Bertrand, N., Dubslaff, C., and Klüppelholz, S. (eds.) Principles of Formal Quantitative Analysis: Essays Dedicated to Christel Baier on the Occasion of Her 60th Birthday. pp. 158–173. Springer Nature Switzerland, Cham (2026). https://doi.org/10.1007/978-3-031-97439-7_7.

3. Shapley, L.S.: Stochastic Games*. Proceedings of the National Academy of Sciences. 39, 1095–1100 (1953). https://doi.org/10.1073/pnas.39.10.1095.

4. Littman, M.L.: Markov games as a framework for multi-agent reinforcement learning, https://www.sciencedirect.com/science/article/pii/B9781558603356500271, (1994). https://doi.org/https://doi.org/10.1016/B978-1-55860-335-6.50027-1.

5. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: PRISM-games 3.0: Stochastic Game Verification with Concurrency, Equilibria and Time. In: Proc. 32nd International Conference on Computer Aided Verification (CAV'20). pp. 475–487. Springer (2020).

6. Zhu, Q., Li, H., Han, Z., Basar, T.: A Stochastic Game Model for Jamming in Multi-Channel Cognitive Radio Systems. In: Proc. ICC'10. pp. 1–6. IEEE (2010).

7. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Automatic Verification of Concurrent Stochastic Systems, https://arxiv.org/abs/2008.04613.

8. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Equilibria-based Probabilistic Model Checking for Concurrent Stochastic Games, https://arxiv.org/abs/1811.07145.

9. Chatterjee, K., de Alfaro, L., Henzinger, T.A.: Strategy improvement for concurrent reachability and turn-based stochastic safety games. Journal of Computer and System Sciences. 79, 640–657 (2013). https://doi.org/https://doi.org/10.1016/j.jcss.2012.12.001.

10. de Alfaro, L., Majumdar, R.: Quantitative solution of omega-regular games. Journal of Computer and System Sciences. 68, 374–397 (2004). https://doi.org/https://doi.org/10.1016/j.jcss.2003.07.009.

11. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Correlated Equilibria and Fairness in Concurrent Stochastic Games, https://arxiv.org/abs/2201.09702.

12. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. Program. Lang. Syst. 16, 1512–1542 (1994). https://doi.org/10.1145/186025.186051.

13. Parker, D.: Implementation of Symbolic Model Checking for Probabilistic Systems, (2002).

14. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Symbolic Verification and Strategy Synthesis for Turn-based Stochastic Games. In: Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday. p. . Springer (2022).

15. Bahar, R., Frohm, E., Gaona, C., Hachtel, G., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of 1993 International Conference on Computer Aided Design (ICCAD). pp. 188–191 (1993). https://doi.org/10.1109/ICCAD.1993.580054.

16. Alfaro, L. de, Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. In: Graf, S. and Schwartzbach, M. (eds.) Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00). pp. 395–410. Springer (2000).

17. Somenzi, F.: CUDD: CU Decision Diagram Package. (2018).

18. Coudert, O., Madre, J.: A unified framework for the formal verification of sequential circuits. In: 1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers. pp. 126–129. IEEE Comput. Soc. Press (1990). https://doi.org/10.1109/ICCAD.1990.129859.

19. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic Model Checking: Advances and Applications. In: Formal System Verification. pp. 73–121. Springer International Publishing (2018). https://doi.org/10.1007/978-3-319-57685-5_3.

20. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: A Model Checker for Stochastic Multi-Player Games. In: Piterman, N. and Smolka, S. (eds.) Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13). pp. 185–191. Springer (2013).

21. Alfaro, L. de, Henzinger, T.A., Kupferman, O.: Concurrent reachability games. Theoretical Computer Science. 386, 188–217 (2007). https://doi.org/10.1016/j.tcs.2007.07.008.

22. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Automated Verification of Concurrent Stochastic Games. In: Proc. 15th International Conference on Quantitative Evaluation of SysTems (QEST'18). pp. 223–239. Springer (2018).

23. Alur, R., Henzinger, T.A., Kupferman, O.: Alternating-time temporal logic. J. ACM. 49, 672–713 (2002). https://doi.org/10.1145/585265.585270.

24. Alfaro, L. de, Henzinger, T.A.: Concurrent Omega-Regular Games. In: Proceedings of the Fifteenth Annual IEEE Symposium on Logic in Computer Science (LICS 2000). pp. 141–154. IEEE Computer Society Press, Santa Barbara, CA, USA (2000).

25. Winnicki, A., Srikant, R.: A New Policy Iteration Algorithm For Reinforcement Learning in Zero-Sum Markov Games, https://arxiv.org/abs/2303.09716.

26. Tripathi, R., Valkanova, E., Anil Kumar, V.: On strategy improvement algorithms for simple stochastic games. Journal of Discrete Algorithms. 9, 263–278 (2011). https://doi.org/https://doi.org/10.1016/j.jda.2011.03.007.

27. Alfaro, L. de, Kwiatkowska, M., Norman, G., Parker, D., Segala, R.: Symbolic Model Checking of Probabilistic Processes Using MTBDDs and the Kronecker Representation. Presented at the (2000). https://doi.org/10.1007/3-540-46419-0_27.

28. Kwiatkowska, M., Norman, G., Parker, D., Santos, G.: Symbolic Verification and Strategy Synthesis for Turn-based Stochastic Games, https://arxiv.org/abs/2211.06141.

29. Hansson, H., Jonsson, B.: A Logic for Reasoning about Time and Reliability. Formal Aspects of Computing. 6, 512–535 (1994).

30. Santos, G.: Automatic Verification and Strategy Synthesis for Zero-sum and Equilibria Properties of Concurrent Stochastic Games, (2021).

31. Bryant, R.E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. ACM Comput. Surv. 24, 293–318 (1992). https://doi.org/10.1145/136035.136043.

32. Clarke, E.M., Fujita, M., McGeer, P.C., McMillan, K., Yang, J.C.-Y., Zhao, X.: Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. (1993). https://doi.org/10.1184/R1/6607613.v1.

33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: Gopalakrishnan, G. and Qadeer, S. (eds.) Proc. 23rd International Conference on Computer Aided Verification (CAV'11). pp. 585–591. Springer (2011).

34. Brenguier, R.: PRALINE: A~Tool for Computing Nash Equilibria in Concurrent Games. In: Sharygina, N. and Veith, H. (eds.) Proceedings of the 23th International Conference on Computer Aided Verification (CAV'13). pp. 890–895. Springer, Saint Petersburg, Russia (2013). https://doi.org/10.1007/978-3-642-39799-8_63.

35. Lemke, C.E., Jr., J.T.H.: Equilibrium points of bimatrix games. SIAM Journal on Applied Mathematics. 12, 413–423 (1964).

36. Hart, S., Mas-Colell, A.: A Simple Adaptive Procedure Leading to Correlated Equilibrium. Econometrica. 68, 1127–1150 (2000). https://doi.org/https://doi.org/10.1111/1468-0262.00153.

37. Aumann, R.J.: Subjectivity and correlation in randomized strategies. Journal of Mathematical Economics. 1, 67–96 (1974). https://doi.org/https://doi.org/10.1016/0304-4068(74)90037-8.

38. Aumann, R.J.: Correlated Equilibrium as an Expression of Bayesian Rationality. Econometrica. 55, 1–18 (1987).

39. Tammelin, O.: Solving Large Imperfect Information Games Using CFR+, https://arxiv.org/abs/1407.5042.

40. Brown, N., Sandholm, T.: Solving Imperfect-Information Games via Discounted Regret Minimization, https://arxiv.org/abs/1809.04040.

41. Fujita, M., Matsunaga, Y., Kakuda, T.: On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In: Proc Eur Conf Des Autom. pp. 50–54. Publ by IEEE (1992).

42. Ishiura, N., Sawada, H., Yajima, S.: Minimization of binary decision diagrams based on exchanges of variables. 1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers. 472–475 (1991).

43. Panda, S., Somenzi, F.: Who are the variables in your neighborhood. In: Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design. pp. 74–77. IEEE Computer Society, San Jose, California, USA (1995).

44. Brenguier, R.: PRALINE: A Tool for Computing Nash Equilibria in Concurrent Games. In: Proc. 25th International Conference on Computer Aided Verification (CAV'13). pp. 890–895. Springer (2013).

45. Zhu, Q., Basar, T.: Dynamic policy-based IDS configuration. In: Proc. 48th IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference. pp. 8600–8605. IEEE (2009).

46. async-profiler development team: async-profiler: Sampling CPU and HEAP profiler for Java, (2026).

47. Dijk, T. van, Pol, J. van de: Sylvan: Multi-Core Decision Diagrams. In: Baier, C. and Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 677–691. Springer Berlin Heidelberg, Berlin, Heidelberg (2015).

48. Husung, N., Dubslaff, C., Hermanns, H., Köhl, M.A.: OxiDD: A Safe, Concurrent, Modular, and Performant Decision Diagram Framework in Rust. In: Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24) (2024). https://doi.org/10.1007/978-3-031-57256-2_13.

49. Zhang, B.H., Anagnostides, I., Sandholm, T.: Scale-Invariant Regret Matching and Online Learning with Optimal Convergence: Bridging Theory and Practice in Zero-Sum Games, https://arxiv.org/abs/2510.04407.

50. He, A.Y., Parker, D.: Robust Verification of Concurrent Stochastic Games, https://arxiv.org/abs/2601.12003.

Appendix

Case Study Prop ID Prop Type	Args	Symbolic		Explicit		Explicit w Caching	
		Cons. Time (s)	Check Time (s)	Cons. Time (s)	Check Time (s)	Cons. Time (s)	Check Time (s)
Jam4 Pr 1 `Pmax=? [F..]`	10	**0.01**	**0.05**	0.47	0.48	0.40	0.39
	23	**0.02**	**0.33**	1.22	4.21	1.23	3.82
Jam6 Pr 1 `Pmax=? [F..]`	10	**0.01**	**0.06**	1.72	1.35	1.77	1.25
	23	**0.02**	**0.42**	8.09	17.77	7.87	16.62
Jam6 Pr 2 `Rmax=? [F..]`	10	**0.01**	**0.37**	1.73	2.01	1.74	1.56
	23	**0.02**	**5.54**	8.04	15.55	8.08	11.22
Aloha Pr 3 `Pmax=? [F..]`	4,4,0.8	**0.12**	**0.66**	1.44	3.05	1.41	3.08
	5,4,0.8	**0.12**	**0.81**	1.96	5.15	1.92	5.03
Aloha Pr 1 `Rmin=? [F..]`	2,2,0.8	**0.06**	1.28	0.15	0.83	0.16	**0.78**
	3,3,0.8	**0.09**	9.40	0.40	**2.60**	0.39	5.93
Robot Pr 1 `Pmax=? [..U..]`	10,0.1	**0.01**	**2.28**	0.51	4.32	0.50	4.11
	13,0.1	**0.02**	**9.21**	1.21	20.82	1.17	19.63
Robot Pr 3 `Rmin=? [F..]`	8,0.1	**0.01**	2.81	0.30	**1.65**	0.28	1.70
	10,0.1	**0.01**	10.00	0.49	4.37	0.56	**3.87**
IDS Pr 1 `Rmin=? [F..]`	250,250	**0.02**	2.09	0.05	3.32	0.05	**1.52**
	500,500	**0.03**	8.13	0.08	12.55	0.07	**5.08**
IDS Pr 2 `Rmin=? [F..]`	250,250	**0.02**	**0.69**	0.05	2.91	0.05	0.96
	500,500	**0.03**	**2.45**	0.07	10.91	0.07	2.60

Case Study Prop ID Prop Type	Args	Symbolic		Explicit		Explicit w Caching	
		Cons. Time (s)	Check Time (s)	Cons. Time (s)	Check Time (s)	Cons. Time (s)	Check Time (s)
	750,750	**0.04**	5.40	0.12	26.79	0.09	**5.03**

Table 9: Comparison of Construction and Checking Times for LP-Based Model Checking Algorithms

Case Study Prop ID Prop Type	Args	Symbolic		Explicit		Explicit w Caching	
		Qual Time (s)	Quant Time (s)	Qual Time (s)	Quant Time (s)	Qual Time (s)	Quant Time (s)
Jam4 Pr 1 `Pmax=? [F..]`	10	**0.00**	**0.05**	0.32	0.15	0.29	0.11
	23	**0.02**	**0.31**	3.20	1.01	3.19	0.63
Jam6 Pr 1 `Pmax=? [F..]`	10	**0.00**	**0.06**	0.91	0.44	0.88	0.37
	23	**0.02**	**0.40**	15.00	2.77	14.66	1.96
Jam6 Pr 2 `Rmax=? [F..]`	10	**0.00**	**0.37**	0.41	1.59	0.40	1.16
	23	**0.00**	**5.54**	3.45	12.10	3.45	7.77
Aloha Pr 3 `Pmax=? [F..]`	4,4,0.8	**0.05**	0.62	2.77	**0.28**	2.75	0.33
	5,4,0.8	**0.05**	0.76	4.62	**0.52**	4.48	0.55
Aloha Pr 1 `Rmin=? [F..]`	2,2,0.8	**0.02**	1.27	0.13	0.70	0.14	**0.64**
	3,3,0.8	**0.03**	9.37	0.41	**2.19**	0.44	5.49
Robot Pr 1 `Pmax=? [..U..]`	10,0.1	**0.15**	2.12	3.61	0.71	3.44	**0.67**
	13,0.1	**0.41**	8.80	18.67	2.15	17.99	**1.64**
	8,0.1	**0.00**	2.81	0.19	**1.46**	0.20	1.50

Case Study **Prop ID** **Prop Type**	**Args**	**Symbolic**		**Explicit**		**Explicit w Caching**	
		Qual Time (s)	**Quant Time (s)**	**Qual Time (s)**	**Quant Time (s)**	**Qual Time (s)**	**Quant Time (s)**
Robot Pr 3 `Rmin=? [F..]`	10,0.1	**0.00**	10.00	0.49	3.87	0.49	**3.39**
IDS Pr 1 `Rmin=? [F..]`	250,250	**0.01**	2.08	0.29	3.03	0.28	**1.24**
	500,500	**0.01**	8.12	0.79	11.76	0.81	**4.27**
IDS Pr 2 `Rmin=? [F..]`	250,250	**0.01**	**0.68**	0.28	2.62	0.28	**0.68**
	500,500	**0.01**	2.44	0.84	10.07	0.85	**1.75**
	750,750	**0.02**	5.38	1.88	24.91	1.63	**3.40**

Table 10: Comparison of Qualitative and Quantitative Times for LP-Based Model Checking Algorithms