July 22, 2022

Bachelor's Thesis

A Model Checker for Game Logic via Parity Games

Eelke Landsaat
eelkelandsaat@student.rug.nl

First supervisor: prof. dr. H. H. Hansen
Second supervisor: prof. dr. J. A. Pérez

University of Groningen, Faculty of Science and Engineering
BSc Computing Science

Abstract

Model checking of logical models is establishing whether a formula is satisfied in a model under certain conditions. Logical models are the groundwork of many software protocols, and ensuring that these protocols work as intended is essential to building reliable applications. Automated model checkers are thus a valuable tool for verifying the correctness of such software protocols.

Game logic is a branch of logic that focuses on strategic interactions between two agents. Although a theoretical foundation for the model checking of game models (models of game logic) already exists, a model checker for them remains absent.

In this thesis, a model checker for game models is implemented through a conversion step to the solving of parity games. To this end, the state of the art around game logic, parity games and model checking is inspected, the procedures of the model checker are discussed, and the implementation is evaluated.

Additionally, an experimental time complexity analysis is performed on the model checker based on existing literature on a theoretical time complexity bound for the model checking problem for game logic.

Acknowledgements

I would like to express my gratitude to my first supervisor, dr. Helle Hvid Hansen, for her flexibility, her swift correspondence, and her guidance regarding both the contents of the project and the process of carrying it out. Her input and feedback both during and outside our meetings have been invaluable for the project.

Additionally, I would like to thank my second supervisor, dr. Jorge A. Perez, for his willingness to evaluate this work and for his flexibility regarding the formal procedure at the start and end of the project.

Contents

1 Introduction and Motivation 5

2 Preliminaries 6
2.1 Game Logic . . . 6
2.2 Parity Games . . . 10
2.2.1 The Evaluation Game . . . 11
2.3 Model Checking . . . 12

3 Program Description 13
3.1 Requirements . . . 13
3.1.1 Must Have . . . 13
3.1.2 Should Have . . . 13
3.1.3 Could Have . . . 13
3.2 Functionality . . . 14
3.3 Implementation . . . 14
3.3.1 Programming Language . . . 14
3.3.2 Parity Game Solver . . . 14
3.3.3 Input Format . . . 15
3.3.4 Process . . . 18
3.3.5 Key Procedures . . . 19
3.3.6 Output Format . . . 21
3.4 Installation and Usage . . . 22

4 Program Testing 23
4.1 Small Inputs . . . 23
4.2 Large Inputs . . . 31

5 Experimental Time Complexity Verification 31
5.1 Theoretical Upper Time Bounds . . . 32
5.2 Upper Time Bound Parity Game Solver . . . 32
5.3 Hypotheses . . . 33
5.4 Experimental Set-Up . . . 34
5.4.1 Input Generation . . . 35
5.4.2 Description Experiment 1 . . . 36
5.4.3 Description Experiment 2 . . . 36
5.4.4 Description Experiment 3 . . . 37
5.4.5 Description Experiment 4 . . . 37
5.4.6 Description Experiment 5 . . . 37
5.4.7 Description Experiment 6 . . . 38
5.4.8 Description Experiment 7 . . . 38
5.5 Results . . . 38
5.5.1 Results Experiment 1 . . . 38
5.5.2 Results Experiment 2 . . . 39
5.5.3 Results Experiment 3 . . . 40
5.5.4 Results Experiment 4 . . . 43
5.5.5 Results Experiment 5 . . . 44
5.5.6 Results Experiment 6 . . . 44
5.5.7 Results Experiment 7 . . . 45

5.6 Conclusion . . . 49

6 Conclusion **49**
6.1 General Conclusions . . . 49
6.2 Future Work . . . 50
6.2.1 Program Modifications . . . 50
6.2.2 Time Complexity Evaluation . . . 50

7 Appendix **54**

A Proofs **54**

B Program Testing **55**
B.1 Small Inputs . . . 55
B.2 Large Input Example . . . 65

C Time Complexity Verification **67**
C.1 Tables of Execution Time Measurements . . . 67

D Source Code (per module) **75**
D.1 ModelChecker . . . 75
D.2 EvalGame . . . 78
D.3 DNNF . . . 80
D.4 Consistency . . . 81
D.5 AbstractSyntax . . . 83
D.6 ConcreteSyntax . . . 84
D.7 AST . . . 85
D.8 Util . . . 88
D.9 ComplexityAnalysis . . . 88

1 Introduction and Motivation

Model checking of logical models backing protocols in software systems is a crucial tool to ensure the correctness of these protocols and in some cases even their safety. What it involves is checking whether certain assertions hold under given circumstances in the logical model. This check can be performed by hand, but having a software tool to perform it effortlessly and without fail is a tremendous luxury for protocol writers to sift out any undesired behaviours. Consider, for example, an elevator. When the call button is pressed on some floor, the elevator should reach that floor, and preferably as soon as possible. More critically, it should not move before the doors are firmly closed, or open the doors in the middle of a ride. A software testing suite could be used to make bugs like these unlikely to occur, but as Edsger Dijkstra [1] famously remarked: *"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."* Using a model checker, one can ensure that bugs like these are absent from the logical model behind the elevator protocol.

Model checkers can be created for various logics (e.g., the modal μ-calculus [2], linear temporal logic [3], computation tree logic (CTL) and [4], the temporal logic of actions [5]). Most notably, to support the analysis of any expression that lies within the expressive power of the logic language, variations of propositional modal logic are used, where the modalities can express information such as probability or time. CTL is one such variation, which shows a series of events in a tree structure, so time flows as the tree is traversed. Although CTL is already effective at showing the properties of processes over time, it is not suitable for reasoning about strategic interactions between two agents. This is where game logic exceeds CTL in expressive power. Using the fixpoint operators from the modal μ-calculus in its semantics for repeated interactions between two players, it makes interaction protocols more intuitive to reason about, and is thus a fitting option for model checking in these situations.

The game logic put forward by Parikh [6] provided a decidable and likely complete logic, but a completeness proof including both the dual operator and the iteration operator [7] was left open (although conjectured by the addition of an axiom). After Berwanger [8] showed in 2003 that game logic covers every finite level of the alternation hierarchy of the modal μ-calculus, it became clear that a proof would be challenging to produce. Sixteen years later, however, Enqvist et al. [9] established a completeness proof of several proof systems (Par, G, CloG, CloM) for game logic, justifying the validity of further work in the area.

In [10], a mapping (further discussed in Section 2.2.1) from the model checking problem (Section 2.3) for game models (Section 2.1) to solving parity games (Section 2.2) is proposed. Parity games are determined two-player games, tools for solving which already exist [11], [12], [13]. Therefore, implementing a model checker for game logic can be achieved by mapping the problem to a representation that can be used as input for such a solver, and mapping its output back to a conclusion for the game model check. Although the semantic mapping already exists, writing a program to perform this mapping has remained open. As such, we attempt to answer the following pair of questions in this work:

1. *"How can the model checking problem for game logic be programmatically converted to a parity game representation for an existing parity game solver?"*

2. *"How can the output of this solver be converted back to a model check result for the game model?"*

The research area that this project lies in within computing science is logic, and more specifically model checking of game logic models (which in turn have a basis in game theory). Its contribution to the field is a model checking tool for game logic, which will directly help two-agent interaction protocol writers to create correct protocols and indirectly help future researchers in the field to more quickly gain an understanding of game logic to use it in their own context. Additionally, an experimental time complexity evaluation is carried out for the tool (Section 5) based on the theoretical bound put forth by Pauly [7], after which the results of the experiments are compared to the theoretical bound.

2 Preliminaries

2.1 Game Logic

Propositional game logic was introduced by Parikh [6] in 1983. It was built on propositional dynamic logic (PDL) for programs developed by Fischer and Ladner [14] (which can be seen as a logic for single-player games) by adding an opponent agent (player 2), which we will refer to as "Demon", with player 1 being "Angel". Game logic is a modal logic with, for our purposes, a single modality, which acquires meaning when paired with a game (we will denote games by the letters α and β). Additionally, games may be described by tests of a formula (we will denote formulas by φ and ψ), giving formulas and games a mutually recursive structure. When the term in consideration may be either a game or a formula, we will use the letter ξ.

Definition 1. (From [10]) *We define the countable set Prop of atomic propositions and the set Gam of atomic games. The sets $\mathcal{F}$ of formulas and $\mathcal{G}$ of game terms of game logic are then defined recursively as follows:*

$$\begin{aligned} &\mathcal{F} \ni \varphi ::= p \in \mathrm{Prop} \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \langle\alpha\rangle\varphi && \textit{where } \alpha \in \mathcal{G} \\ &\mathcal{G} \ni \alpha ::= g \in \mathrm{Gam} \mid \alpha^d \mid \alpha \cup \alpha \mid \alpha \cap \alpha \mid \alpha;\alpha \mid \alpha^* \mid \alpha^\times \mid \varphi? \mid \varphi! && \textit{where } \varphi \in \mathcal{F} \end{aligned}$$

The logical operators negation ($\neg$), conjunction ($\wedge$), and disjunction ($\vee$) are defined as usual. $\langle\rangle$ is the diamond modality. The expression $\langle\alpha\rangle\varphi$ means: "Angel has a strategy in the game α to ensure that it ends in a state in which φ is true". It should be noted here that, since game logic describes zero-sum games, Angel has a strategy to ensure φ if and only if Demon does not have a strategy to ensure $\neg\varphi$. As such, the expression $\neg\langle\alpha\rangle\neg\varphi$ says that Demon has a strategy in α to ensure φ.

α^d is the dual game of α, in which Angel and Demon swap roles. I.e., $\langle\alpha^d\rangle\varphi \equiv$ "Angel has a strategy in α^d to ensure φ" $\equiv$ "Demon has a strategy in α to ensure φ".

The angelic choice $\alpha \cup \beta$ describes the game in which Angel gets to choose which game is played out of α and β. The demonic choice $\alpha \cap \beta$ describes the game in which Demon chooses between α and β.

The composition $\alpha;\beta$ describes the game in which α and β are played sequentially, in that order.

The angelic iteration α^* describes the game in which Angel chooses how many times α is played in the range $[0, \rightarrow\rangle$. In a demonic iteration $\alpha^\times$, Demon chooses the number of times α is played in the range $[0, \rightarrow\rangle$. In both cases, the number of times that α is played must be finite. Additionally, the decision whether to play α another time at any point in the sequence may be determined with consideration of the outcomes of the previous games, so the number of times that it is played in total is not predetermined. As a consequence, $\langle\alpha^*\rangle\varphi$ means that there exists some finite number of times that α can be played sequentially for Angel to ensure φ. $\langle\alpha^\times\rangle\varphi$ means that Angel can sustain a situation in which φ is true regardless of the number of times α is played. Both angelic iterations and demonic iterations are fixpoint operators and may be referred to as such.

In the angelic test game $\varphi?$, Angel loses immediately if φ is false. I.e., if $\langle\varphi?\rangle\psi \wedge \neg\varphi$, then Angel does not have a strategy to ensure ψ, as she immediately loses. The demonic test $\varphi!$ is the game in which Demon loses immediately if φ is false. I.e., in $\langle\varphi!\rangle\psi$, Angel has a strategy to ensure ψ if either φ is true or ψ is true.

A formal semantics for game logic can be defined using monotone neighbourhood frames [15].

Definition 2. (From [10]) *Let S be a set. We denote by $\mathcal{M}(S)$ the set of up-closed subsets of $\mathcal{P}(S)$, i.e., $\mathcal{M}(S) = \{N \subseteq \mathcal{P}(S) \mid \forall U, U' : U \in N, U \subseteq U' \Rightarrow U' \in N\}$. A monotone neighbourhood frame on S is a function $f : S \to \mathcal{M}(S)$. We denote by $\mathrm{MF}(S)$ the set of all monotone neighbourhood frames on S.*

Given a monotone neighbourhood frame $f \in \mathrm{MF}(S)$ and a state s in S, $f(s)$ is the set of neighbourhoods that Angel can force (what it means for Angel to force a neighbourhood is explained later). A neighbourhood is no more than a set of states, which we will frequently denote by the letter U. What it means to be monotone for a neighbourhood frame f with domain S is that all the sets of neighbourhoods that it maps to are closed under superset with respect to S. E.g., if $S = \{s_1, s_2\}$, $f(s)$ cannot equal $\{\{s_1\}\}$, as all the supersets of

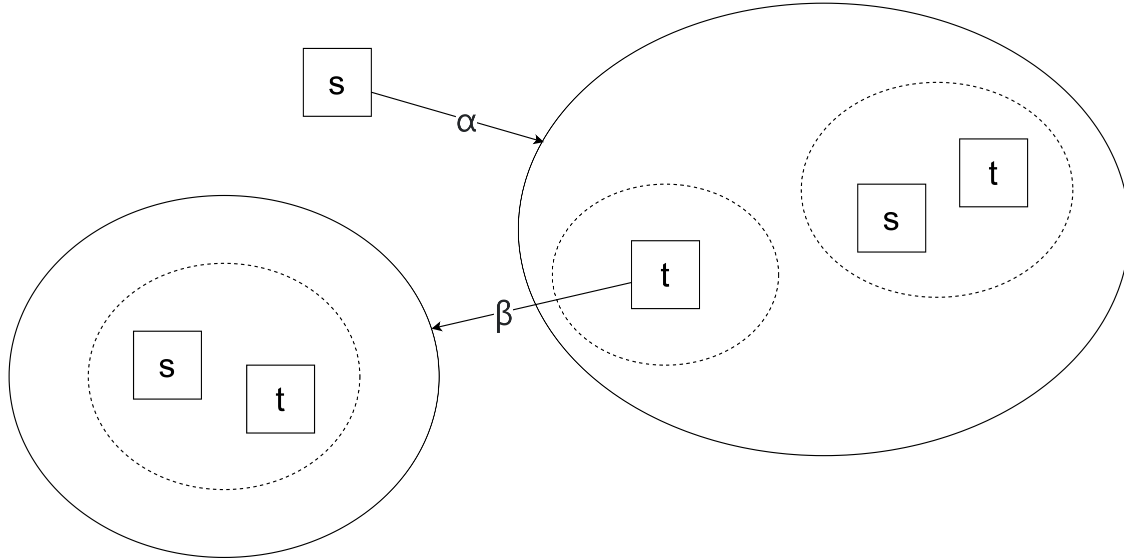


Figure 2.1: Graphical representation of a part of a simple game model. States are indicated by squares, sets are indicated by ellipses. Neighbourhoods have a dotted outline. Arrows indicate a mapping through the monotone neighbourhood frame corresponding to the game of which the name is specified on the arrow.

$\{s_1\}$, being only $\{s_1, s_2\}$ in this case, has to be included in the image as well. Later we will see that it is not necessary for a neighbourhood frame to have the monotonic property for our purpose.

Definition 3. (Game operations, from [10]) *Let $f, g, f_1, f_2 \in \mathrm{MF}(S)$ be monotone neighbourhood frames. We define*

- *the unit frame η_S by: $U \in \eta_S(s)$ iff $s \in U$ for $s \in S$ and $U \subseteq S$*
- *the composition $f_1; f_2$ by:*

$$U \in (f_1; f_2)(s) \text{ iff } \{s' \in S \mid U \in f_2(s')\} \in f_1(s) \text{ for } s \in S \text{ and } U \subseteq S$$

- *the Angelic choice and Demonic choice between f_1 and f_2 by:*

$$(f_1 \cup f_2)(s) = f_1(s) \cup f_2(s) \quad (f_1 \cap f_2)(s) = f_1(s) \cap f_2(s), \quad \text{for } s \in S$$

- *the dual f^d by: $U \in f^d(s)$ iff $S \setminus U \notin f(s)$ for $s \in S$ and $U \subseteq S$*
- *the angelic iteration by $f^* := \mathrm{LFP}(A_f)$*
- *the demonic iteration by $f^\times := \mathrm{GFP}(D_f)$*

where $\mathrm{LFP}(A_f)$ and $\mathrm{GFP}(D_f)$ are the least and greatest fixed points of the maps

$$\begin{array}{rcl} \mathrm{A}_f : \mathrm{MF}(S) & \to & \mathrm{MF}(S) \\ g & \mapsto & \eta_S \cup (f; g) \end{array} \qquad \begin{array}{rcl} \mathrm{D}_f : \mathrm{MF}(S) & \to & \mathrm{MF}(S) \\ g & \mapsto & \eta_S \cap (f; g) \end{array}$$

Before we continue to the definition of a game model, it is in our favour to gain an intuition for the way they are used in game logic. A game model consists of a number of states, a number of atomic propositions and a number of atomic games. For each proposition, there is a set of states, its truth set, in which it holds.

Each atomic game g in a game model $\mathcal{M}$ is bound to a monotone neighbourhood frame $\langle\!|g|\!\rangle_{\mathcal{M}}$, telling us which state transitions can take place in g. In any state s, if e.g., $\langle\!|g|\!\rangle_{\mathcal{M}}(s) = \{U_1, ..., U_n\}$, and the game g is played, then Angel chooses (forces) a neighbourhood $U_i = \{s_1, ...s_m\}, 1 \leq i \leq n$, after which Demon chooses a state $s_j, 1 \leq j \leq m$ to transition to (see Figure 2.1 for visual reference). Notably, a neighbourhood can also be the empty set.

Definition 4. (From [10]) *A game model is a triple $\mathcal{M} = (S, \gamma, \Upsilon)$ where S is a set of states, $\gamma : \text{Gam} \to \text{MF}(S)$ is a* Gam*-indexed collection of monotone neighbourhood frames, which provides an interpretation of atomic games, and $\Upsilon : \text{Prop} \to \mathcal{P}(S)$ is a valuation of atomic propositions. For $\varphi \in \mathcal{F}$ and $\alpha \in \mathcal{G}$ we define the semantics $[\![\varphi]\!]_{\mathcal{M}} \subseteq S$ and $\langle\!|\alpha|\!\rangle_{\mathcal{M}} \in \text{MF}(S)$ by induction on the term structure:*

$$\begin{array}{rlrl}
[\![p]\!]_{\mathbb{S}} := \Upsilon(p) \quad \textit{for } p \in \mathsf{Prop} & & [\![\neg\varphi]\!]_{\mathbb{S}} & := S \setminus [\![\varphi]\!]_{\mathbb{S}} \\
[\![\varphi_1 \vee \varphi_2]\!]_{\mathbb{S}} := [\![\varphi_1]\!]_{\mathbb{S}} \cup [\![\varphi_2]\!]_{\mathbb{S}} & & [\![\varphi_1 \wedge \varphi_2]\!]_{\mathbb{S}} & := [\![\varphi_1]\!]_{\mathbb{S}} \cap [\![\varphi_2]\!]_{\mathbb{S}} \\
[\![\langle\alpha\rangle\varphi]\!]_{\mathbb{S}} := \{s \in S \mid [\![\varphi]\!]_{\mathbb{S}} \in \langle\!|\alpha|\!\rangle_{\mathbb{S}}(s)\} & & \langle\!|\alpha;\beta|\!\rangle_{\mathbb{S}} & := \langle\!|\alpha|\!\rangle_{\mathbb{S}} \,;\, \langle\!|\beta|\!\rangle_{\mathbb{S}} \\
\langle\!|g|\!\rangle_{\mathbb{S}} := \gamma(g) \ \textit{for } g \in \mathsf{Gam} & & \langle\!|\alpha^d|\!\rangle_{\mathbb{S}} & := (\langle\!|\alpha|\!\rangle_{\mathbb{S}})^d \\
\langle\!|\alpha \cup \beta|\!\rangle_{\mathbb{S}} := \langle\!|\alpha|\!\rangle_{\mathbb{S}} \cup \langle\!|\beta|\!\rangle_{\mathbb{S}} & & \langle\!|\alpha \cap \beta|\!\rangle_{\mathbb{S}} & := \langle\!|\alpha|\!\rangle_{\mathbb{S}} \cap \langle\!|\beta|\!\rangle_{\mathbb{S}} \\
\langle\!|\alpha^*|\!\rangle_{\mathbb{S}} := (\langle\!|\alpha|\!\rangle_{\mathbb{S}})^* & & \langle\!|\alpha^\times|\!\rangle_{\mathbb{S}} & := (\langle\!|\alpha|\!\rangle_{\mathbb{S}})^\times \\
\langle\!|\psi?|\!\rangle_{\mathbb{S}} := \lambda x. \begin{cases} \eta_S(x) & \textit{if } x \in [\![\psi]\!]_{\mathbb{S}} \\ \emptyset & \textit{otherwise.} \end{cases} & & \langle\!|\psi!|\!\rangle_{\mathbb{S}} & := \lambda x. \begin{cases} \eta_S(x) & \textit{if } x \notin [\![\psi]\!]_{\mathbb{S}} \\ \mathcal{P}S & \textit{otherwise.} \end{cases}
\end{array}$$

The semantics of Definition 4 applies to all neighbourhood models, not just to monotone ones. We say that two game logic formulas φ and ψ are equivalent if for all game models $\mathcal{M}$, $[\![\varphi]\!]_{\mathcal{M}} = [\![\psi]\!]_{\mathcal{M}}$. Similarly, we say that two games α and β are equivalent if for all $\mathcal{M}$, $\langle\!|\alpha|\!\rangle_{\mathcal{M}} = \langle\!|\beta|\!\rangle_{\mathcal{M}}$. When it is clear from the context which game model is being considered, we will abbreviate $\langle\!|\alpha|\!\rangle_{\mathcal{M}}$ to $\langle\!|\alpha|\!\rangle$ and $[\![\varphi]\!]_{\mathcal{M}}$ to $[\![\varphi]\!]$.

Definition 5. (From [10]) *A formula $\varphi \in \mathcal{F}$, respectively game term $\alpha \in \mathcal{G}$, is in dual and negation normal form* (DNNF) *if dual is only applied to atomic games and negations occur only in front of atomic propositions. We denote by $\text{nf}(\varphi)$ the equivalent of φ in* DNNF *and by $\text{nf}(\alpha)$ the equivalent of α in* DNNF*. We denote by $\mathcal{F}_{\text{DNNF}}$ the set of formulas in* DNNF*, and by $\mathcal{G}_{\text{DNNF}}$ the set of game terms in* DNNF*.*

Note that any formula or game term can be converted to its equivalent in DNNF [10], for example, the formula $\varphi = \neg p_2 \wedge \langle (g_1^* \cup ((p_2 \vee p_1)!)^d)^\times \rangle p_3$ can be converted to its DNNF equivalent as follows:

$$\begin{array}{lll}
 & \neg p_2 \wedge \langle (g_1^* \cup ((p_2 \vee p_1)!)^d)^\times \rangle p_3 & \\
\equiv & & ((\psi!)^d \equiv (\neg\psi)?) \quad \forall\psi \\
 & \neg p_2 \wedge \langle (g_1^* \cup (\neg(p_2 \vee p_1))?)^\times \rangle p_3 & \\
\equiv & & \text{Distributivity negation over disjunction.} \\
 & \neg p_2 \wedge \langle (g_1^* \cup (\neg p_2 \wedge \neg p_1)?)^\times \rangle p_3 &
\end{array}$$

$$\text{nf}(\varphi) = \neg p_2 \wedge \langle (g_1^* \cup (\neg p_2 \wedge \neg p_1)?)^\times \rangle p_3$$

We continue with some definitions that will be useful to categorize game models and formulas later. We start with a definition of the size of a game model, intuitively equal to the number of states in the model, added to the number of non-unique states that can be mapped to through the neighbourhood frames in the model.

Definition 6. (From [16], rephrased) *For a game model $\mathcal{M} = (S, \gamma, \Upsilon)$, we denote by $|\mathcal{M}|$ the cardinality or size of $\mathcal{M}$, given by*

$$|S| + \sum_{g \in \text{Dom}(\gamma)} \sum_{s \in \text{Dom}(\langle\!|g|\!\rangle)} \sum_{U \in \langle\!|g|\!\rangle(s)} |U|.$$

Definition 7. *For a game logic formula or game term, we define its cardinality or size recursively as follows (with $p \in \text{Prop}; g \in \text{Gam}; \varphi, \psi \in \mathcal{F}; \alpha, \beta \in \mathcal{G}$):*

$$\begin{aligned}
|p| &= 1 \\
|\neg\varphi| &= 1 + |\varphi| \\
|\varphi \wedge \psi| &= 1 + |\varphi| + |\psi| \\
|\varphi \vee \psi| &= 1 + |\varphi| + |\psi| \\
|\langle\alpha\rangle\varphi| &= 1 + |\alpha| + |\varphi| \\
|g| &= 1 \\
|\alpha^d| &= 1 + |\alpha| \\
|\alpha \cup \beta| &= 1 + |\alpha| + |\beta| \\
|\alpha \cap \beta| &= 1 + |\alpha| + |\beta| \\
|\alpha; \beta| &= 1 + |\alpha| + |\beta| \\
|\alpha^*| &= 1 + |\alpha| \\
|\alpha^\times| &= 1 + |\alpha| \\
|\varphi?| &= 1 + |\varphi| \\
|\varphi!| &= 1 + |\varphi|
\end{aligned}$$

The alternation depth $\text{ad}(\varphi)$ of a game logic formula or game term ξ is an important property in regards to its complexity [7]. It can be thought of as the deepest nesting of alternating angelic and demonic iterators in any subgame of ξ. When the nesting of a game continues in a formula φ, by way of an angelic or demonic test, the alternation depth count of any subgames of φ starts at 0 again.

Definition 8. (From [16]) *The alternation depth of a game logic formula φ in dual normal form is defined as follows:*

$$\begin{aligned}
\text{ad}(p) &= 0 \text{ for } p \in \text{Prop} \\
\text{ad}(\varphi \vee \psi) &= \max(\text{ad}(\varphi), \text{ad}(\psi)) \\
\text{ad}(\neg\varphi) &= \text{ad}(\varphi) \\
\text{ad}(\langle\alpha\rangle\varphi) &= \max(\text{ad}(\alpha), \text{ad}(\varphi)) \\
\text{ad}(g) &= 0 \text{ for } g \in \text{Gam} \\
\text{ad}(\varphi?) &= \text{ad}(\varphi) \\
\text{ad}\left(\alpha^d\right) &= \text{ad}(\alpha) \\
\text{ad}(\alpha \cup \beta) = \text{ad}(\alpha \cap \beta) &= \max(\text{ad}(\alpha), \text{ad}(\beta)) \\
\text{ad}(\alpha; \beta) &= \max(\text{ad}(\alpha), \text{ad}(\beta)) \\
\text{ad}\left(\alpha^*\right) &= \max\left(1, \text{ad}(\alpha), 1 + \text{ad}\left(\alpha_1^\times\right), \ldots, 1 + \text{ad}\left(\alpha_n^\times\right)\right) \\
&\quad \text{where } \alpha_i^\times \text{is a subgame of } \alpha \text{ not in the scope of ? or !} \\
\text{ad}\left(\alpha^\times\right) &= \max\left(1, \text{ad}(\alpha), 1 + \text{ad}\left(\alpha_1^*\right), \ldots, 1 + \text{ad}\left(\alpha_n^*\right)\right) \\
&\quad \text{where } \alpha_i^* \text{ is a subgame of } \alpha \text{ not in the scope of ? or !}
\end{aligned}$$

Since the set of formulas in DNNF is a subset of the formulas in dual normal form, we can use this definition of alternation depth for our purposes.

It will become clear in section 5 that it can be valuable to keep the size of a game model $\mathcal{M} = (S, \gamma, \Upsilon)$ to a minimum. To this end, the non-monotonic core $\text{nmc}(\mathcal{M})$ of $\mathcal{M}$ may be used.

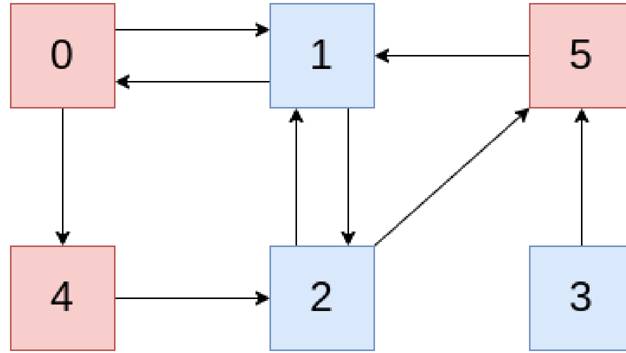


Figure 2.2: A simple parity game arena. The colour of a position specifies which player it belongs to. Priorities are indicated by numbers on the positions. Note that the priorities do not have to form a singular interval of N, but can be distributed (e.g., using the priorities $\{3, 6, 24\}$ on a game board with 3 positions).

Definition 9. *We define the non-monotonic core* $\mathrm{nmc}(\mathcal{M})$ *of the game model* $\mathcal{M} = (S, \gamma, \Upsilon)$ *by construction, following a short sequence of steps:*

1. *Let* $\mathcal{M}' := \mathcal{M}$.
2. $\forall g \in \mathrm{Dom}(\gamma), \forall t \in \mathrm{Dom}(g), \forall U \in \langle\!| g |\!\rangle t$*: if any subset of* U *is an element of* $\langle\!| g |\!\rangle t$*,* U *is removed from* $\langle\!| g |\!\rangle t$ *in* $\mathcal{M}'$.
3. $\mathrm{nmc}(\mathcal{M}) := \mathcal{M}'$.

We add a theorem that makes the non-monotonic core powerful in the context of model checking.

Theorem 1. *For any game model* $\mathcal{M} = (S, \gamma, \Upsilon)$*:* $\mathcal{M}, s \models \varphi$ *iff* $\mathrm{nmc}(\mathcal{M}), s \models \varphi$*,* $\forall s \in S, \forall \varphi \in \mathcal{F}$.

2.2 Parity Games

Parity games (see also [10]) are positionally determined, zero-sum, two-player graph games. Note that some definitions used here do not apply to parity games in general. For a more elaborate account, see [17].

Parity games are played in an arena $\mathbb{B}$ (see Figure 2.2) with a game board B, a finite set of positions. Every position $b \in B$ is owned by exactly one of two players, referred to as Abelard ($\forall$ for short) and Eloise ($\exists$ for short). We refer to the set of positions owned by Abelard as $B_\forall$ and the set of positions owned by Eloise as $B_\exists$. As such, $B_\forall$ and $B_\exists$ together form a partition of B. We denote by $\mathrm{P}(b)$ the owner of position b. The arena comes with a binary relation $E \subseteq B \times B$, describing the edges of the graph, which define the successors of each position. We denote by $E[b] \subseteq B$ the set of successors of position b.

A parity game is played by moving a pebble across the edges of $\mathbb{B}$, where the owner of the position which the pebble is currently on makes the next move. Note that a player can move the pebble to a position owned by themselves if the edges allow, permitting them to move several times in sequence without intervention of the opponent. A play starts at an initial position b_0, after which, given optimal play, the winner of the game is fixed. A full play is either finite or infinite, in each of which the winning condition differs. A play that is not full is called a partial play. If a full play is finite, then there is some position b such that $E[b] = \emptyset$, where the play ends. We refer to this situation as the owner of b getting stuck. In this case, the player who gets stuck loses the play. If the full play $\Pi = \{b_0 b_1 ... b_n ...\} \in B^\omega$ is infinite, we require a parity function to determine the winner. A parity function $\Omega : B \to \mathbb{N}$ maps every position on the game board to a natural number, referred to as its priority. Let m be the highest priority of the positions that occur infinitely often in Π. The infinite play Π is then won by $\begin{cases} \text{Eloise} & \text{if } m \text{ is even} \\ \text{Abelard} & \text{if } m \text{ is odd} \end{cases}$.

A strategy for a player is a partial function $f : B \to B$ that maps positions to one of their successors. If a position b has no successors or is owned by the other player, $f(b)$ is undefined. A play $\Pi = \{b_1...b_n...\} \in B^* \cup B^\omega$ follows the strategy f if $f(b_i) = b_{i+1}$ for all $b \in B$ on which f is defined. A winning strategy for player i from position b is a strategy that ensures a win for player i, regardless of the moves of the other player. Importantly, a player can have a winning strategy from b even if b is not owned by them (e.g., if the opponent only has one move from position b, but the principle applies to more intricate cases as well). Since parity games are positionally determined, there is a winning strategy for exactly one of the players from each position on its game board. We denote the set of winning positions for player i in the parity game $\mathbb{B}$ by $\mathrm{Win}_i(\mathbb{B})$. Solving $\mathbb{B}$ locally for a position b means determining whether $b \in \mathrm{Win}_\exists(\mathbb{B})$ or $b \in \mathrm{Win}_\forall(\mathbb{B})$. Solving $\mathbb{B}$ globally means determining whether $b \in \mathrm{Win}_\exists(\mathbb{B})$ or $b \in \mathrm{Win}_\forall(\mathbb{B})$ for all $b \in B$.

2.2.1 The Evaluation Game

The evaluation game due to Hansen et al. [10] is a parity game that can be constructed based on a game model and a formula in DNNF, which is particularly useful for model checking (see Section 2.3) in game logic due to Theorem 2. The positions of an evaluation game are pairs (s, φ) or (U, φ). Before introducing the definition of the evaluation game (Definition 12), we introduce some terminology that serves as its foundation.

Definition 10. (From [10]) *We let $\lhd \subseteq (\mathcal{F} \cup \mathcal{G})^2$ be the subterm relation on formulas and game terms, i.e., $\xi_1 \lhd \xi_2$ if either $\xi_1 = \xi_2$ or ξ_1 is a proper subterm of ξ_2.*

ξ_1 being a proper subterm of ξ_2 can be thought of as ξ_1 being nested in ξ_2. E.g., if $\alpha = \langle\beta\rangle\varphi$, then both β and φ are proper subterms of α.

Definition 11. (From [10]) *For a term $\xi \in \mathcal{F} \cup \mathcal{G}$ we let $\mathrm{Fix}(\xi) := \{\alpha^* \mid \alpha \in \mathcal{G}, \alpha^* \lhd \xi\} \cup \{\alpha^\times \mid \alpha \in \mathcal{G}, \alpha^\times \lhd \xi\}$ be the set of all fixpoint games that are subterms of ξ. A parity function for a formula φ in* DNNF *is a partial map $\Omega : \mathrm{Fix}(\varphi) \to \omega$ such that*

1. *$\alpha_1 \lhd \alpha_2$ implies $\Omega(\alpha_1) < \Omega(\alpha_2)$ for all $\alpha_1, \alpha_2 \in \mathrm{Fix}(\varphi)$ with $\alpha_1 \neq \alpha_2$, and*
2. *for all $\alpha \in \mathrm{Fix}(\varphi)$, $\Omega(\alpha)$ is even iff $\alpha = \beta^\times$ is a demonic iteration.*

We define the canonical parity function $\Omega_{\mathrm{can}}(\alpha^*) = 2n + 1$ and $\Omega_{\mathrm{can}}(\alpha^\times) = 2n$ where $n = \#\mathrm{Fix}(\alpha^*)$ and $n = \#\mathrm{Fix}(\alpha^\times)$, respectively. The canonical parity function formalises the fact that any fixpoint operator dominates any other fixpoint operator in its scope.

Definition 12. (From [10]) *Let $\mathcal{M} = (S, \gamma, \Upsilon)$ be a game model, let $\varphi \in \mathcal{F}$ be a formula in* DNNF *and let $\Omega : \mathrm{Fix}(\varphi) \to \omega$ be a parity function for φ. We define the evaluation game $\mathcal{E}(\mathcal{M}, \varphi)$ as the parity game with the game board specified in Figure 2.3 and the parity function $\Omega_\mathcal{E}$ given by*

$$\Omega_\mathcal{E} := \begin{cases} \Omega(\alpha) & \text{if } b = (x, \langle\alpha\rangle\psi) \text{ for some } \alpha \in \mathrm{Fix}(\varphi) \\ 0 & \text{otherwise.} \end{cases}$$

We finish with the theorem that is the key to the usefulness of the evaluation game to model checking for game logic.

Theorem 2. (From [10]) *Let $\mathcal{M} = (S, \gamma, \Upsilon)$ be a game model and consider the game $\mathcal{E} = \mathcal{E}(\mathcal{M}, \varphi)$ for some $\varphi \in \mathcal{F}$. Then for all positions (s, ψ) in $\mathcal{E}$ we have $(s, \psi) \in \mathrm{Win}_\exists(\mathcal{E})$ iff $\mathcal{M}, s \models \psi$.*

Formula Part

Position b	P(b)	Moves E[b]
$(s,p), s \in \Upsilon(p)$	$\forall$	$\emptyset$
$(s,p), s \notin \Upsilon(p)$	$\exists$	$\emptyset$
$(s,\neg p), s \in \Upsilon(p)$	$\exists$	$\emptyset$
$(s,\neg p), s \notin \Upsilon(p)$	$\forall$	$\emptyset$
$(s,\varphi \wedge \psi)$	$\forall$	$\{(s,\varphi),(s,\psi)\}$
$(s,\varphi \vee \psi)$	$\exists$	$\{(s,\varphi),(s,\psi)\}$
$(s,\langle g\rangle\varphi)$	$\exists$	$\{(U,\langle g\rangle\varphi) \mid U \in \llparenthesis g \rrparenthesis(s)\}$
$(U,\langle g\rangle\varphi)$	$\forall$	$\{(s,\varphi) \mid s \in U\}$
$(s,\langle g^d\rangle\varphi)$	$\forall$	$\{(U,\langle g^d\rangle\varphi) \mid U \in \llparenthesis g \rrparenthesis(s)\}$
$(U,\langle g^d\rangle\varphi)$	$\exists$	$\{(s,\varphi) \mid s \in U\}$

Game Part

Position b	P(b)	Moves E[b]
$(s,\langle\alpha\,;\beta\rangle\varphi)$	$\star$	$\{(s,\langle\alpha\rangle\langle\beta\rangle\varphi)\}$
$(s,\langle\alpha\cup\beta\rangle\varphi)$	$\star$	$\{(s,\langle\alpha\rangle\varphi\vee\langle\beta\rangle\varphi)\}$
$(s,\langle\alpha\cap\beta\rangle\varphi)$	$\star$	$\{(s,\langle\alpha\rangle\varphi\wedge\langle\beta\rangle\varphi)\}$
$(s,\langle\alpha^*\rangle\varphi)$	$\star$	$\{(s,\varphi\vee\langle\alpha\rangle\langle\alpha^*\rangle\varphi)\}$
$(s,\langle\alpha^\times\rangle\varphi)$	$\star$	$\{(s,\varphi\wedge\langle\alpha\rangle\langle\alpha^\times\rangle\varphi)\}$
$(s,\langle\psi?\rangle\varphi)$	$\star$	$\{(s,\psi\wedge\varphi)\}$
$(s,\langle\psi!\rangle\varphi)$	$\star$	$\{(s,\psi\vee\varphi)\}$

Figure 2.3: Definition of the evaluation game board due to Hansen et al. [10] A star in the P(b) column indicates that it is irrelevant who owns b, as there is exactly one possible move.

2.3 Model Checking

In many logics, it is valuable to check whether a formula is true under certain conditions, described by a model. Specifically, given a model $\mathcal{N}$ and a formula f, the problem at hand is to check whether $\mathcal{N} \models f$. This is referred to as the model checking problem.

The latest overview for game logic, by Pauly and Parikh [16], published in 2003, describes developments around the model checking problem for game logic. Local model checking for game logic involves determining whether a formula $\varphi \in \mathcal{F}$ is true at a given state in a given game model. I.e., for a game model $\mathcal{M}$, a state s, and a formula φ, a model check is a check whether $\mathcal{M}, s \models \varphi$. Model checking can also be performed globally, i.e., locally checking for all states in the model at once, yielding one result per state. Note that this is different from checking if φ is true in all states in the model. Model checking has made greater developments outside the realm of game logic than inside it, being developed further in other modal fixpoint logics.

The basis of modal fixpoint logics is the modal mu-calculus, model checking for which has been a widely researched field since the late 1980s. Stirling and Walker [2] were first to present a local model checker for the modal μ-calculus. Following this, different variations, such as a linear-time algorithm for the alternation-free fragment of the modal μ-calculus by Cleaveland and Steffen [18]; an incremental algorithm for the same fragment by Sokolsky and Smolka [19]; faster checking for the full calculus with alternating fixpoints by Cleaveland, Klein and Steffen [20] and finally an overview of methods until 2018 by Bradfield and Walukiewicz [21].

CTL has had a plethora of papers dedicated to its model checking as well. Okawa and Yoneda [22] proposed a symbolic model checker for CTL in 1997. Later, model checking on variations on CTL was further explored, such as that on quantum CTL (QCTL) by Baltazar, Chadha and Mateus [23] and on full hybrid CTL* by Kernberger and Lange [24]. Moreover, different techniques were examined, an example of which is model checking based on possibility measures [25], [26].

To perform a model check on a game model $\mathcal{M} = (S, \gamma, \Upsilon)$ and a formula φ in game logic via a parity game (the evaluation game), we have to undertake a sequence of steps, depending on whether the model check is local or global. In the local model checking case for a state s, we

1. construct the evaluation game $\mathcal{E} = \mathcal{E}(\mathcal{M}, \varphi)$;
2. solve $\mathcal{E}$ locally at (s, φ);

3. conclude that $\begin{cases} \text{M, s} \models \varphi & \text{if (s, } \varphi) \in \text{Win}_\exists(\mathcal{E}) \\ \text{M, s} \not\models \varphi & \text{otherwise} \end{cases}$.

In the global model checking case, we

1. construct the evaluation game $\mathcal{E} = \mathcal{E}(\mathcal{M}, \varphi)$;
2. solve $\mathcal{E}$ globally;
3. conclude that $\begin{cases} \text{M, t} \models \varphi & \text{if (t, } \varphi) \in \text{Win}_\exists(\mathcal{E}) \\ \text{M, t} \not\models \varphi & \text{otherwise} \end{cases} \forall t \in S$.

3 Program Description

We consider the requirements for the model checker, the functionality that was implemented to meet these requirements, the implementation of the program at a high level, and the code of the program and how to use it.

3.1 Requirements

To list the requirements in a comprehensive manner, we order them by priority. To this end, we use the categories "Must have", "Should have", and "Could have". When, alongside these, a "Will not have" category is included as well, this practice is known as the MoSCoW method. Since the "Will not have" category is usually used by software developers to indicate to their client that a requirement is too much to include in a product, which is not the setting of this work, this category is excluded here. Whether a requirement was met is indicated by a checkbox, where a check signifies success and a minus signifies that progress has been made towards the requirement, but it is not yet completed. The priority of the requirements is based on the moment at which their addition was put forward, ranging from before the proposal until the end of the project period, as well as on their significance to the project result. All the requirements listed are functional, i.e., they influence how the program behaves and do not affect properties like execution speed, compatibility, or reliability.

3.1.1 Must Have

- ☑ R-1.1: Local game logic model checking by mapping to an evaluation game and solving it with a parity game solver.

3.1.2 Should Have

- ☑ R-2.1: Global game logic model checking by mapping to an evaluation game and solving it with a parity game solver.

3.1.3 Could Have

- ☑ R-3.1: A selection of parity game solvers to choose from by command- line arguments for solving the evaluation game corresponding to the input.
- ⊟ R-3.2: An input format for the formula that uses game logic syntax.
- ⊟ R-3.3: An indication of the state transitions for a winning strategy in the (game model, formula) pair in the output.

3.2 Functionality

The functionality of the model checker and the completed requirements align in a one-to-one fashion. Requirements R-1.1 and R-2.1 allow the user to locally or globally model check a game logic formula, which was the principal aim of the project and has been implemented. Requirement R-3.1 allows the user to choose a parity game solver that best fits their use case for optimal performance (a guideline for which can be found in Section 3.3.2). This, too, has been implemented. For requirement R-3.3, the model checker currently shows one winning state transition for the winning player for each state, but a complete walk-through of the winning strategy is not always possible due to the current implementation. More details on this can be found in Section 3.3.5.

3.3 Implementation

3.3.1 Programming Language

For the implementation of the model checker, the meta-programming language Rascal[1] was chosen. Rascal provides a parser generator based on a developer-defined syntax (see Appendix D.6) in its standard library, keeping code complexity for input interpretation low and flexibility high. Additionally, the conversion of game logic formulas to DNNF, the conversion of a game model and a formula to the corresponding evaluation game, and the conversion from a concrete syntax tree (CST) to an abstract syntax tree (AST) all require a considerable amount of case distinction, for which the pattern matching Rascal provides is particularly suitable. Furthermore, Rascal is an interpreted language that runs on a Java virtual machine (JVM), making it highly compatible across different operating systems. Although other programming languages provide a subset of these functionalities as well (Haskell[2] has excellent pattern matching, the combination of C with Flex[3] and Bison [4] provides flexible lexer and parser generation), no other language integrates them natively to the extent Rascal does.

3.3.2 Parity Game Solver

To allow for requirement R-3.3, integrating various different parity game solver programs in the model checker would entail customizing the input generation and the output interpretation for each one, greatly increasing the code complexity. Although a module could be developed to perform a translation from a unified parity game solver format to the specific input that each solver requires, it is more elegant to have a separate program perform this step. This program, integrated with a collection of parity game solvers, already exists in the form of the PGSolver project [11], the input format for which is shown in Figure 3.1. With 3 local solvers and 28 global solvers, it provides a substantial number of options to choose from to optimize the solution to a specific problem. More solvers, e.g., the Oink collection [12] or the solver by Fearnley [13] could be added as well, but they have been disregarded for now.

To accommodate model checking for users who are not interested in the underlying implementation, a default parity game solver is chosen that is used when no specific one is requested. This solver should be as little ‘opinionated’ as possible, to ensure it performs reasonably well for all inputs. The big step solver due to Schewe [27] meets this condition the best out of the solvers in the PGSolver collection. Its time complexity is shown in Equation 1.

$$\mathcal{O}(e \cdot n^{\frac{1}{3}d}) \tag{1}$$

Here, e is the number of edges on the game board, n is the number of positions, and d is the number of unique priority values.

[1] `https://www.rascal-mpl.org/`
[2] `https://www.haskell.org/`
[3] `https://github.com/westes/flex`
[4] `https://www.gnu.org/software/bison/`

$$
\begin{aligned}
\langle \textit{parity_game} \rangle &::= [\texttt{parity}\ \langle \textit{identifier} \rangle\ ;]\ \langle \textit{node_spec} \rangle^{+} \\
\langle \textit{node_spec} \rangle &::= \langle \textit{identifier} \rangle\ \langle \textit{priority} \rangle\ \langle \textit{owner} \rangle\ \langle \textit{successors} \rangle\ [\langle \textit{name} \rangle]\ ; \\
\langle \textit{identifier} \rangle &::= \mathbb{N} \\
\langle \textit{priority} \rangle &::= \mathbb{N} \\
\langle \textit{owner} \rangle &::= \texttt{0} \mid \texttt{1} \\
\langle \textit{successors} \rangle &::= \langle \textit{identifier} \rangle\ (,\ \langle \textit{identifier} \rangle)^{*} \\
\langle \textit{name} \rangle &::= \texttt{"}\ (\text{ any ASCII string not containing `}\texttt{"}\text{'})\ \texttt{"}
\end{aligned}
$$

Figure 3.1: The parity game input grammar in extended Backus–Naur form for the PGSolver collection of parity game solvers [11].

3.3.3 Input Format

As the goal of the model checker is to transform a game model and a game logic formula into an evaluation game, it is convenient to have the input syntax of the model checker be of similar form as the input grammar for the parity game solver (Figure 3.1). The input grammar for the model checker can be found in Figure 3.2. For consistency, it is formulated in the same extended Backus-Naur form as the grammar for PGSolver. Note that any text ⟨*between_angle_brackets_like_this*⟩ describes a nonterminal, while any other text is a literal, with the exception of parentheses around parts that are followed by $^+$ or * or that contain a pipe symbol | (e.g., in "(example1 | example2)*", the parentheses are not part of the literal characters, while in "not(example)" they are). This input format was chosen over one which imitates game logic formula syntax more closely, as it eliminates any ambiguity due to associativity or operator precedence. This decision comes at the cost of readability for the input. To combat this, game logic syntax could be integrated into the model checker as well, but this addition has been set aside for now.

Using the definition of Figure 3.2, instances of ⟨*input*⟩ can be used as input for the model checker. An example is shown in Listing 1, the corresponding game model and formula for which in game logic notation are described below.

		Definitions	Notes
⟨*input*⟩	::=	⟨*game_model*⟩ ⟨*formula*⟩ [⟨*state*⟩]	⟨*state*⟩ required for local checking.
⟨*game_model*⟩	::=	model	
		⟨*state_defs*⟩ ; ⟨*neighbourhood_func*⟩*	
		end model	
⟨*state_defs*⟩	::=	⟨*state_def*⟩ (, ⟨*state_def*⟩)*	
⟨*state_def*⟩	::=	⟨*state*⟩ ⟨*proposition*⟩*	
⟨*state*⟩	::=	⟨*identifier*⟩	
⟨*proposition*⟩	::=	⟨*identifier*⟩	
⟨*neighbourhood_func*⟩	::=	⟨*atomic_game*⟩ : (⟨*state_map*⟩ ;)$^+$ end func	
⟨*atomic_game*⟩	::=	⟨*identifier*⟩	
⟨*state_map*⟩	::=	⟨*state*⟩ -> ⟨*neighbourhoods*⟩	
⟨*neighbourhoods*⟩	::=	⟨*neighbourhood*⟩ (, ⟨*neighbourhood*⟩)*	
⟨*neighbourhood*⟩	::=	⟨*state*⟩$^+$ $\mid$ empty	
⟨*formula*⟩	::=	⟨*proposition*⟩	
		$\mid$ not(⟨*formula*⟩)	$\neg\varphi$
		$\mid$ and(⟨*formula*⟩ , ⟨*formula*⟩)	$\varphi \wedge \psi$
		$\mid$ or(⟨*formula*⟩ , ⟨*formula*⟩)	$\varphi \vee \psi$
		$\mid$ strat(⟨*game*⟩ , ⟨*formula*⟩)	$\langle\alpha\rangle\varphi$
⟨*game*⟩	::=	⟨*atomic_game*⟩	
		$\mid$ dual(⟨*game*⟩)	α^d
		$\mid$ ang_choice(⟨*game*⟩ , ⟨*game*⟩)	$\alpha \cup \beta$
		$\mid$ dem_choice(⟨*game*⟩ , ⟨*game*⟩)	$\alpha \cap \beta$
		$\mid$ seq(⟨*game*⟩ , ⟨*game*⟩)	$\alpha \; ; \; \beta$
		$\mid$ ang_iter(⟨*game*⟩)	α^*
		$\mid$ dem_iter(⟨*game*⟩)	$\alpha^\times$
		$\mid$ ang_test(⟨*formula*⟩)	$\varphi?$
		$\mid$ dem_test(⟨*formula*⟩)	$\varphi!$
⟨*identifier*⟩	::=	(⟨*letter*⟩ $\mid$ _) (⟨*letter*⟩ $\mid$ ⟨*digit*⟩ $\mid$ _)*	
⟨*letter*⟩	::=	a-z $\mid$ A-Z	
⟨*digit*⟩	::=	0-9	

Figure 3.2: Input grammar for the model checker in extended Backus-Naur form.

$\mathcal{M} = (S, \gamma, \Upsilon)$

$S = \{s_1, s_2, s_3, s_4\}$

γ is defined by:

Game g	State s	$\langle\!\langle g \rangle\!\rangle(s)$
g_1	s_1	$\{\{s_2, s_3\}, \{s_1, s_2, s_3\}\}$
g_2	s_1	$\{\emptyset, \{s_1\}, \{s_2\}, \{s_3\}, \{s_1, s_2\}, \{s_1, s_3\}, \{s_2, s_3\}, \{s_1, s_2, s_3\}\}$
	s_2	$\{\{s_2\}, \{s_1, s_2\}, \{s_2, s_3\}, \{s_1, s_2, s_3\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_2, s_4\}$
p_2	$\{s_1, s_4\}$
p_3	$\{s_4\}$

$\varphi = \neg p_2 \wedge \langle (g_1^* \cup ((p_2 \vee p_1)!)^d)^\times \rangle p_3$

Here, γ and Υ are represented by a table for readability. The table for γ contains, for every atomic game g in $\mathcal{M}$, all the states that have a mapping in the neighbourhood frame corresponding to g, along with their image in this neighbourhood frame. The table for Υ contains every proposition in the model along with the set of states in which it is true.

Note that in this example, the model contains all the supersets of neighbourhoods in the mappings of its neighbourhood frames, meeting the monotonicity constraint of game models. This, however, is not required, as the constructed evaluation games from both a game model with a monotonic neighbourhood frame and its non-monotonic core produce equivalent evaluation games with respect to winning positions (Theorem 1, Theorem 2).

```
model
s1 p1 p2,
s2 p1,
s3,
s4 p3 p1 p2;
g1:
  s1 -> s1 s2 s3, s3 s2;
end func
g2:
  s2 -> empty, s1, s2, s3, s1 s2, s1 s3, s2 s3, s1 s2 s3;
  s1 -> s2, s1 s2, s2 s3, s1 s2 s3;
end func
end model
and(not(p2), strat(dem_iter(ang_choice(ang_iter(g1), dual(dem_test(or(p2, p1))
    ↪ ))), p3))
s3
```

Listing 1: An example of a well-formed input for the model checker. Using this input, a local model check is performed on the state named s3.

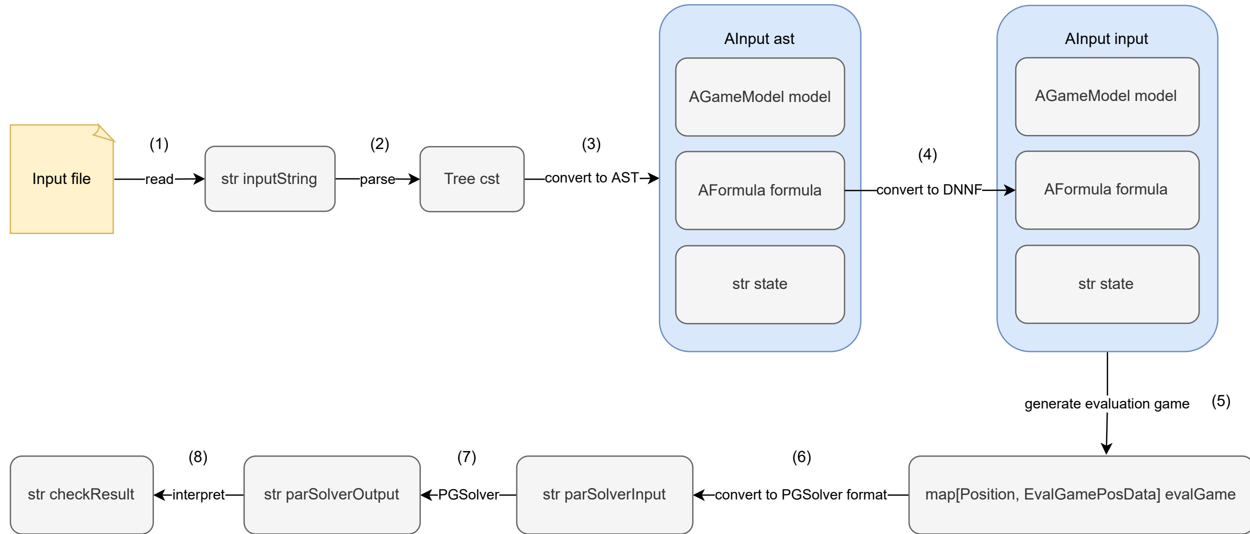


Figure 3.3: High-level overview of the processing steps of the model checker. Rounded rectangles represent instances of data types which the program uses. Arrows represent a processing step (a function call, a sequence of function calls, or the execution of an external program). All data types shown are present in the actual program.

3.3.4 Process

When an input is given to the model checker, it performs a local or global model check through a sequence of processing steps. These steps are shown in Figure 3.3 and work as follows:

(1) The input file is read into the program and its content is converted to a string. The format that the input file must have is defined in module `ConcreteSyntax`, which can be found in Appendix D.6.

(2) The input string is converted to a CST. This conversion results in an instance of the built-in Rascal data type `Tree`. A CST contains a large amount of metadata that is not necessary for the model check (e.g., the location of each character of the input file on the machine).

(3) The CST is converted to an AST, which is stripped of all the unnecessary metadata. The data type of the AST is called `AInput` (abstract input) and can be found in module `AbstractSyntax` located in Appendix D.5. The conversion is performed by module `AST` in Appendix D.7

(4) The formula to be checked is converted to DNNF in preparation of the evaluation game generation. The conversion function is defined in module `DNNF` in Appendix D.3.

(5) The evaluation game is generated and represented by a collection of parity game positions. This collection is represented internally by a hash table (Rascal data type: `map`) which maps instances of `Position` to instances of `EvalGamePosData`. These data types and the functions to generate an evaluation game are defined in the module `EvalGame` in Appendix D.2. A `Position` closely imitates the definition of an evaluation game position in [10], as it consists of a state or neighbourhood, along with a formula. An instance of `EvalGamePosData` contains the data that is required by PGSolver to model the evaluation game (an integer identifier, a priority, the owner, and a list of identifiers of neighbouring positions). By mapping `Positions` to `EvalGamePosData`, we keep track of the way the evaluation game corresponds to the output that PGSolver will produce, which is necessary for the

output interpretation (step 8). Additionally, the use of a hash table provides lookup of elements by key in constant time, allowing the evaluation game generation procedure to quickly determine whether a new node is required for an edge to connect to or an edge to an existing node should be added.

(6) All the `evalGamePosData` elements from the evaluation game representation are converted to a PGSolver input string of the form defined in Figure 3.1.

(7) The requested parity game solver from the PGSolver collection is executed with the generated input string.

(8) The output of PGSolver is interpreted using the mapping created in step (5) and converted to a model check result as described in Section 3.3.6.

In between steps (3) and (4), a check takes place whether the AST contains any contradictions with itself. This check is performed by the `Consistency` module in Appendix D.4. The AST is checked for

- references to states that were not defined in the state definitions of the game model (before the neighbourhood frames);
- references to propositions in the formula that were not defined in the game model;
- references to atomic games in the formula that were not defined in the game model;
- the existence of more than one neighbourhood frame corresponding to the same atomic game;
- the existence of more than one mapping of a state to a set of neighbourhoods within the same neighbourhood frame;
- the existence of more than one state definition with the same name.

If one or more of these situations occur in the AST, the program prints a descriptive error message for each of them and terminates.

3.3.5 Key Procedures

We consider the pivotal procedures of the model checker. The most crucial part of the tool is the translation step from a game model and a formula to a parity game representation that is accepted by PGSolver. This representation requires four elements per position on the game board: an identifier (a natural number), a priority (a natural number), an owner (0 or 1, which we assign to the players Eloise and Abelard respectively), and a list of identifiers of successors. Algorithm 1 (GENERATEBOARD) describes how this translation is performed in the model checker. Here, the `PosData` data type is used (correspondingly `EvalGamePosData` in the source code in Appendix D.2). This data type has the four elements described above, with names `id`, `prio`, `owner`, and `succ`, where `succ` is a set of natural numbers. The algorithm returns a position-indexed relation to `PosData` elements, after which only a conversion to a string representation according to the grammar of Figure 3.1 is required before PGSolver can be executed.

Algorithm 1 repeatedly makes procedure calls to REACHABLEPOSITIONS (Algorithm 2), which adds to F all those positions recursively reachable from b in accordance with Figure 2.3 and increases i_{next} accordingly. Algorithm 1 and 2 together ensure that the position identifiers are as low as possible, as PGSolver will generate redundant positions when this is not the case [11].

Any parity function which adheres to the conditions set out in Definition 11 can be used to assign priorities to fixpoint operators in Algorithm 2. We use the canonical parity function here, as it is easily computed and adheres to the aforementioned conditions. A description of the implementation is described in Algorithm 3 (PARITY).

Algorithm 1 GENERATEBOARD: Generate all evaluation game positions for a model check (corresponds to `evalGamePositions`, Appendix D.2). Return a relation from positions to `PosData`.

Require: $\mathcal{M} = (S, \gamma, \Upsilon)$: A game model.
Require: φ: A game logic formula.
Require: $S \supseteq S_{check}$: A list of states to perform the model check for (equal to S for a global check, only contains one state for a local model check).

1: $F \leftarrow \emptyset$ ▷ The relation mapping positions to `PosData`.
2: $i_{next} \leftarrow 0$ ▷ The highest `id` currently in use by a position.
3: **for all** $s \in S_{check}$ **do**
4:     $b \leftarrow (s, \varphi)$
5:     **if** $b \notin \text{Dom}(F)$ **then**
6:         REACHABLEPOSITIONS$(b, i_{next}, F, \mathcal{M})$ ▷ Parameters are passed by reference.
7:     **end if**
8: **end for**
9: **return** F

Algorithm 2 REACHABLEPOSITIONS: Generate all evaluation game positions reachable from a particular position b and store them in F (corresponds to `registerPositions`, Appendix D.2).

Require: $b = (s, \varphi)$: The position to start from.
Require: $i_{next} \in \mathbb{N}$: The highest `id` currently in use by a position.
Require: F: The relation containing the positions which have been generated thus far, excluding b.
Require: $\mathcal{M} = (S, \gamma, \Upsilon)$: A game model, invisibly needed for E and $\Omega_{\mathcal{E}}$.

1: $N \leftarrow E[b]$ ▷ Neighbours (moves) of b, defined in Figure 2.3.
2: $d_b \leftarrow$ empty `PosData` instance
3: d_b.`id` $\leftarrow i_{next}$
4: d_b.`prio` $\leftarrow \Omega_{\mathcal{E}}(b)$ ▷ Definition 11 is used, substituting Ω_{can} for Ω.
5: d_b.`owner` $\leftarrow$ owner(b) ▷ As defined in Figure 2.3.
6: d_b.`succ` $\leftarrow \emptyset$
7: $F[b] \leftarrow d_b$
8: $i_{next} \leftarrow i_{next} + 1$
9: **for all** $n \in N$ **do**
10:     **if** $n \notin \text{Dom}(F)$ **then**
11:         Add i_{next} to $F[b]$.`succ`.
12:         REACHABLEPOSITIONS$(n, i_{next}, F, \mathcal{M})$ ▷ Parameters are passed by reference.
13:     **else**
14:         Add $F[n]$.`id` to $F[b]$.`succ`.
15:     **end if**
16: **end for**

Algorithm 3 PARITY: Determine the priority of an evaluation game position (corresponds to `parity`, Appendix D.2).

Require: $b = (s, \varphi)$ or (U, φ): The position to determine the priority for.

1: **if** b is of the form $(_, \langle\alpha\rangle_)$ **then** ▷ Where _ can be anything which fits in its position and $\alpha \in \mathcal{G}$.
2: **if** α is an angelic iteration **then**
3: **return** $2 \cdot \#\,\mathrm{Fix}(\alpha) + 1$
4: **end if**
5: **if** α is a demonic iteration **then**
6: **return** $2 \cdot \#\,\mathrm{Fix}(\alpha)$
7: **end if**
8: **end if**
9: **return** 0

This implementation of the parity function has the disadvantage that it has to traverse all the subterms of φ to calculate $\#\,\mathrm{Fix}(\alpha)$ every time the procedure is called. This could be mitigated by using an altered data structure for φ (see Section 6.2). For the sake of better code clarity, the model checker uses a data structure that imitates actual game logic formulas and games as closely as possible for now.

In addition to the conversion of a game model and a formula to a parity game representation, the conversion of the PGSolver output to a model check result is rather important as well, though much more straightforward. PGSolver consecutively prints the position identifiers in which player 0 has a winning strategy and those in which player 1 has a winning strategy. Since we assigned player 0 to be Eloise, we can conclude that, for a model check of a state s in a game model $\mathcal{M}$ and a formula φ: $\mathcal{M}, s \models \varphi$ iff the identifier corresponding to (s, φ) is in the list of position identifiers from which Eloise has a winning strategy (Theorem 2).

Besides the winning positions for each player, the PGSolver output also contains winning strategies for some of the positions in the parity game when it is solved globally. This output is used to accommodate for the functionality corresponding to requirement R-3.3. When globally model checking with a game model $\mathcal{M}$ and a formula φ, the model checker looks for the identifiers corresponding to the position (s, φ) in the PGSolver output for every state s in $\mathcal{M}$. The transitions from state to neighbourhood are reported in the model check result accordingly as described in Section 3.3.6. The reason that the model checker does not perform this conversion starting from the PGSolver output, i.e., look through all position identifiers for ones that correspond to a position $(_, \varphi)$, with _ being either a state or a neighbourhood, is that PGSolver (empirically) does not seem to report any transitions from a position that contains a neighbourhood to a position that contains a state. This is also the reason why the functionality for R-3.3 is currently considered incomplete. A possible solution to this issue is proposed in Section 6.2.

3.3.6 Output Format

The output of the model checker takes on a different form depending on whether the check is local or global. In the local case, the output has the following form:

```
In state "<state>", the formula
  "<formula>"
is <result>.
```

Here, `<state>` is the state for which the model check was performed, `<formula>` is the formula that was checked, and `<result>` is the result of the check ("`true`" or "`false`").

The output for a global model check is more elaborate. It looks as follows:

```
The formula
  "<formula>"
is true in states
  {<true_states>}
and false in states
  {<false_states>}
with winning moves
  <winning_moves>
```

Here, `<formula>` has the same meaning as for local checks, `<true_states>` is the list of states in which `<formula>` is true, separated by commas and each state enclosed by double quotes, `<false_states>` is the list of states in which `<formula>` is false, separated by commas and each state enclosed by double quotes, and `<winning_moves>` is a list of winning moves, one for each state in the game model, separated by line breaks. One such winning move has one of the following forms:

```
<state> -> <neighbourhood>
```

or

```
<state> -> stay
```

In the former case, the winning player has a winning strategy that involves choosing the neighbourhood `<neighbourhood>` when they are in the state `<state>`. In the latter case, the winning player does not have any state transitions starting from state `<state>` or never reaches it. Examples of model check outputs can be found in Appendix B.1.

3.4 Installation and Usage

The complete annotated code of the model checker can be found in Appendix D. The entry point is the function `main` in module `ModelChecker` in Appendix D.1.

To run the program, a Rascal installation[5], either the command line REPL or the Eclipse plugin, is required, as well as an installation of PGSolver[6]. Installing the model checker using the command line REPL can be performed as follows:

1. Install a Java JDK 1.8 or higher[7].
2. Create a root folder for the model checker to be inside and navigate into this root folder.
3. Create a folder named "src" and place all the source code files inside it.
4. Place the installation of PGSolver in the root folder (this should consist of a folder named "pgsolver").
5. Create an empty folder named input files. This is where input files are read from.
6. Create a folder named "META-INF" and navigate into it.
7. Create a file named "RASCAL.MF" and populate it with the following string:

[5] `https://www.rascal-mpl.org/start/`
[6] `https://github.com/tcsprojects/pgsolver`
[7] `https://www.oracle.com/java/technologies/downloads/`

```
Manifest-Version: 0.0.1
Source: src
Main-Module: Plugin
Main-Function: main
Courses: courses
Project-Name: GameModelChecker
```

8. Place the installation of the Rascal command line REPL (called “rascal-shell-stable.jar” or similar) in the root folder.

Now, the model checker is installed. It can be executed by typing “java -Xmx`<n>`G -Xss32m -jar rascal-shell-stable.jar `<input_file>`” on a command line in the root folder. Here, `<n>` is the number of gigabytes of memory which is allocated to the program. Usually, 1 gigabyte is sufficient. `<input_file>` is the name of the file containing the input. Optionally, the flag “-solver” can be added, followed by a valid parity game solver name in PGSolver. For local model checking, these solvers are “modelchecker”, “stratimprloc2”, and “stratimprlocal”. For global model checking, these solvers are “bigstep”, “dominiondec”, “external_solver”, “external_solver_univ”, “fpiter”, “genetic”, “guessstrategy”, “modelchecker”, “optstratimprov”, “policyiter”, “prioprom”, “priopromdel”, “priopromdeluniv”, “priopromplus”, “priopromplusuniv”, “priopromrec”, “priopromrecuniv”, “priopromuniv”, “recursive”, “satsolve”, “smallprog”, “stratimprdisc”, “stratimprloc2”, “stratimprlocal”, “stratimprove”, “stratimprsat”, “succinctsmallprog”, and “viasat”. The properties of these solvers can be found via Friedmann and Lange [11].

Executing the model checker in this way takes some time to start the JDK and to generate parsers every time. If consecutive model checks of several input files are desired, it is faster to start the interactive Rascal interpreter using the command “java -Xmx`<n>`G -Xss32m -jar rascal-shell-stable.jar”, followed by importing the library (“import ModelChecker;”) and typing the command “main(["`<input_file>`"]);” in the interpreter for each model check. Using this method, the exposed function for reporting the key parameters regarding the execution time with the input file can also be used by typing “import ComplexityAnalysis;” followed by “reportComplexityParams(["`<input_file>`"]);”. If command line flags are desired using this method, they can be added in the list argument of the main function. E.g., “main(["input_file.txt", "-solver", "modelchecker"]);” would be a valid function call to perform a model check on the model and formula described in the file “input_file.txt” with the parity game solver called “modelchecker”.

4 Program Testing

Testing of the model checker was performed by evaluating whether it produces the right output for various trivial test-cases, followed by tests with randomly generated large input files to see if these give rise to any errors or exceptions. The tests were carried out on a machine with an Intel Core i7-9750H processor at 2.6 GHz with 4 GB of memory allocated, running on Ubuntu 18.04. The expected output was observed in all tests.

4.1 Small Inputs

The set of small inputs consists of one file per formula and game type (see Definition 1). This makes for a total of 13 tests (the diamond modality does not require its own test, as it would be equivalent to the one for atomic games). In each of these tests, a small varying game model is used. Only global model checks are included here, as their local counterparts produce equivalent, but less informative results. Below follows a description of each small input in game logic notation with its result, along with a reference to its corresponding input and output of the model checker (Appendix B.1). No distinction is made between the expected result and the actual result, as these are identical for every test.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: atomic proposition

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_0	$\{s_0, s_3, s_5, s_7\}$
p_1	$\{s_1, s_4\}$
p_2	$\{s_1, s_2\}$
p_3	$\{s_1, s_3\}$
p_4	$\{s_4, s_5\}$
p_5	$\{s_5\}$
p_6	$\{s_4\}$
p_7	$\{s_4, s_7, s_8\}$

$\varphi = p_0$

Result: φ is true in s_0, s_3, s_5, and s_7.

The input and output corresponding to this test can be found in Listing 2 and Listing 3, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: negation

$S = \{s_1, s_2\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1\}$

$\varphi = \neg p_1$

Result: φ is only true in s_2.

The input and output corresponding to this test can be found in Listing 4 and Listing 5, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: conjunction

$S = \{s_1, s_2, s_3\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_3\}$
p_2	$\{s_3\}$

$\varphi = p_1 \wedge p_2$

Result: φ is only true in s_3.

The input and output corresponding to this test can be found in Listing 6 and Listing 7, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: disjunction

$S = \{s_1, s_2, s_3\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_3\}$
p_2	$\{s_3\}$

$\varphi = p_1 \vee p_2$

Result: φ is true in s_1 and s_3.

The input and output corresponding to this test can be found in Listing 8 and Listing 9, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: atomic game

$S = \{s_1, s_2\}$

γ is defined by:

Game g	State s	$\langle\!\langle g \rangle\!\rangle(s)$
g_1	s_1	$\{\{s_1, s_2\}\}$
	s_2	$\{\{s_2\}, \{s_1, s_2\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_2\}$
p_2	$\{s_2\}$

$\varphi = \langle g_1 \rangle p_2$

Result: φ is only true in s_2.

The input and output corresponding to this test can be found in Listing 10 and Listing 11, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: dual game

$S = \{s_1, s_2\}$

γ is defined by:

Game g	State s	$\langle\!\langle g \rangle\!\rangle(s)$
g_1	s_1	$\{\emptyset, \{s_1, s_2\}\}$
	s_2	$\{\{s_1\}\{s_1, s_2\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1\}$
p_2	$\{s_2\}$

$\varphi = \langle g_1^d \rangle p_2$

Result: φ is false in all states.

The input and output corresponding to this test can be found in Listing 12 and Listing 13, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: angelic choice

$S = \{s_1, s_2, s_3\}$

γ is defined by:

Game g	State s	$\langle\!\langle g \rangle\!\rangle(s)$
g_1	s_1	$\{\{s_2, s_3\}\}$
	s_2	$\{s_2, s_3\}$
g_2	s_1	$\{\{s_2, s_3\}\}$
	s_3	$\{\{s_1\}, \{s_2\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_3\}$
p_2	$\{s_2\}$
p_3	$\{s_3\}$

$\varphi = \langle g_1 \cup g_2 \rangle p_1$

Result: φ is only true in s_3.

The input and output corresponding to this test can be found in Listing 14 and Listing 15, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: demonic choice

$S = \{s_1, s_2, s_3\}$

γ is defined by:

Game g	State s	$\langle\!\lvert g \rvert\!\rangle(s)$
g_1	s_1	$\{\{s_2\}, \{s_3\}\}$
	s_2	$\{\{s_2, s_3\}\}$
	s_3	$\{\{s_1, s_2\}\}$
g_2	s_1	$\{\{s_2, s_3\}\}$
	s_3	$\{\{s_1\}, \{s_2\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_2\}$
p_2	$\{s_3\}$
p_3	$\{s_2\}$

$\varphi = \langle g_1 \cap g_2 \rangle p_1$

Result: φ is only true in s_3.

The input and output corresponding to this test can be found in Listing 16 and Listing 17, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: composition

$S = \{s_1, s_2, s_3\}$

γ is defined by:

Game g	State s	$\langle\!\lvert g \rvert\!\rangle(s)$
g_1	s_1	$\{\{s_2, s_3\}\}$
	s_2	$\{\{s_1, s_2\}\}$
g_2	s_1	$\{\{s_1\}\}$
	s_2	$\{\{s_1\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1, s_2\}$
p_2	$\{s_2\}$

$\varphi = \langle g_1 ; g_2 \rangle p_1$

Result: φ is only true in s_2.

The input and output corresponding to this test can be found in Listing 18 and Listing 19, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: angelic iteration

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$

γ is defined by (note the absence of a mapping for s_7):

Game g	State s	$\langle\!|g|\!\rangle(s)$
g_0	s_0	$\{\{s_1\}\}$
	s_1	$\{\{s_2\}\}$
	s_2	$\{\{s_3\}\}$
	s_3	$\{\{s_4\}\}$
	s_4	$\{\{s_5\}\}$
	s_5	$\{\{s_6\}\}$
	s_6	$\{\{s_7\}\}$
	s_8	$\{\{s_9\}\}$
	s_9	$\{\{s_0\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_0	$\{s_0\}$
p_1	$\{s_1\}$
p_2	$\{s_2\}$
p_3	$\{s_3\}$
p_4	$\{s_4\}$
p_5	$\{s_5\}$
p_6	$\{s_6\}$
p_7	$\{s_7\}$
p_8	$\{s_8\}$
p_9	$\{s_9\}$

$\varphi = \langle g_0^* \rangle p_1$

Result: φ is true in the states s_0, s_1, s_8, and s_9.

The input and output corresponding to this test can be found in Listing 20 and Listing 21, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: demonic iteration

$S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9\}$

γ is defined by:

Game g	State s	$\langle\!\langle g \rangle\!\rangle(s)$
g_0	s_0	$\{\{s_1\}\}$
	s_1	$\{\{s_2\}\}$
	s_2	$\{\{s_3\}\}$
	s_3	$\{\{s_4\}\}$
	s_4	$\{\{s_5\}\}$
	s_5	$\{\{s_0\}\}$
	s_6	$\{\{s_7\}\}$
	s_7	$\{\{s_8\}\}$
	s_8	$\{\{s_9\}\}$
	s_9	$\{\{s_6\}\}$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_0	$\{s_0, s_1, s_2, s_3, s_4, s_5, s_7, s_8, s_9\}$

$\varphi = \langle g_0^{\times} \rangle p_0$

Result: φ is true in the states s_0, s_1, s_2, s_3, s_4, and s_5.

The input and output corresponding to this test can be found in Listing 22 and Listing 23, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: angelic test

$S = \{s_1, s_2, s_3\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_1\}$
p_2	$\{s_1, s_2\}$

$\varphi = \langle p_1 ? \rangle p_2$

Result: φ is only true in s_1.

The input and output corresponding to this test can be found in Listing 24 and Listing 25, repectively.

$\mathcal{M} = (S, \gamma, \Upsilon)$ Test: demonic test

$S = \{s_1, s_2, s_3\}$

$\gamma = \emptyset$

Υ is defined by:

Proposition p	$\Upsilon(p)$
p_1	$\{s_3\}$
p_2	$\{s_2, s_3\}$
p_3	$\{s_1\}$

$\varphi = \langle p_2! \rangle p_1$

Result: φ is true in s_2 and s_3.

The input and output corresponding to this test can be found in Listing 26 and Listing 27, repectively.

4.2 Large Inputs

Since the large input sets are too sizeable for a full description of them to be included here, only a description of the parameters relevant for the running time of the model checker ($|\mathcal{M}|, |S|, |\varphi|, \text{ad}(\varphi)$ see Table 1) is given. Input sets were generated using the method described in Section 5.4.1.

When creating large input files, two methods could be considered: keeping all parameters constant, while varying one at a time, or increasing all parameters at the same time. Since the former of these is utilized in Section 5, we will restrict our focus to the latter method here. To be able to execute the model checker with very large input files within a reasonable time frame, we require a parity game solver with a running time that balances out the running time of the model checker as the parameters increase. The way in which the input parameters of the model checker relate to the running time of a parity game solver is explained in Section 5.2. We will use the big step algorithm by Schewe [27] here, as it does not perform poorly with respect to any parameter in particular. Its time complexity is shown in Equation 1. To be able to execute the large inputs within a reasonable time frame, the alternation depth is kept to 1, as high values drastically increase the running time of the model checker. Tests with higher alternation depths can be found in Section 5.5, where the time complexity of the model checker is analyzed.

With $\text{ad}(\varphi) = 1$, the triple $(|\mathcal{M}|, |S|, |\varphi|)$ was varied in the range $[(10, 3, 20), (2000, 600, 4000)]$ with step size $(10, 3, 20)$. The largest input file in this set was processed by the model checker in 15 minutes and 16 seconds. For reference of the approximate form of a generated input file, the actual input with $|\mathcal{M}| = 100, |S| = 30$, and $|\varphi| = 200$ can be found in Appendix B.2. The model checker did not generate any errors or exceptions when executed with one of the large inputs.

5 Experimental Time Complexity Verification

This section presents a collection of experiments performed with the model checker regarding its time complexity with varying parameters. The results of these experiments are then compared to the theoretical analysis by Pauly [7], who set forth two theoretical upper time bounds on global model checking of game models. Conclusions are drawn regarding the efficiency of the tool and the accuracy of the theoretical time bounds.

5.1 Theoretical Upper Time Bounds

Pauly [7] theorized that global game model checking can be performed within the upper time bound described in Equation 2, with $\mathcal{M}$ the game model and φ the game logic formula for which to perform a model check. Table 1 can be used for swift reference of the meaning of each of the mentioned parameters.

$$\mathcal{O}(|\mathcal{M}|^{\mathrm{ad}(\varphi)+1} \times |\varphi|) \tag{2}$$

In addition to the rough upper bound of Equation 2, Pauly also provided a more precise bound, which is usually much lower, given that $|S|$ is usually smaller than $|\mathcal{M}|$. The more accurate bound can be found in Equation 3.

$$\mathcal{O}(|S|^{\mathrm{ad}(\varphi)} \times |\mathcal{M}| \times |\varphi|) \tag{3}$$

As the reasoning behind the time bounds falls largely outside the scope of this work, the reader is referred to [7, Theorem 6.21] and the corresponding proof for reference.

5.2 Upper Time Bound Parity Game Solver

As the upper time bounds given by Pauly concern global model checking, the experiments here are carried out with the global setting of the model checker. The default global parity game solver, the big step algorithm due to Schewe [27], is used for all experiments, as it does not perform particularly weakly with respect to one parameter when compared to any of the other options in the PGSolver collection. Additionally, using the same parity game solver for all experiments provides some consistency amongst the results, the absence of which may lead to wrong conclusions when comparing the plots. The upper time complexity bound of the big step algorithm can be found in Equation 1, which is repeated below for easy reference. Here, e represents the number of edges in the parity game, n the number of nodes, and d the number of unique priorities that are present.

$$\mathcal{O}(e \cdot n^{\frac{1}{3}d})$$

To make the parity game solver bound meaningful in the context of the model checker, we should clarify which input variables for the model checker influence which parameters in the bound. This information can be extracted from Figure 2.3. Given a game logic formula in DNNF, the number of edges e

1. increases linearly with the number of diamond operators that do not work directly on an atomic game or the dual of one;

2. increases exponentially with base 2 with the sum of the number of disjunctions and the number of conjunctions in the formula to be checked;

3. multiplies by the number of states that can be mapped to by an atomic game g for each subterm of the form $\langle g \rangle \varphi$ or $\langle g^d \rangle \varphi$.

This means that e is large for formulas with many conjunctions or disjunctions or for formulas with many diamond operators in combination with atomic games that map to many large neighbourhoods (e.g., when the neighbourhood frames used meet the monotonic property).

The number of nodes n increases with the same parameters and the same relations as e. However, n is strictly bound by e, as new edges can be formed to existing nodes, whereas no new nodes can be formed without also creating a new edge.

The number of different priorities d is equal to $1 + \#\,\mathrm{Fix}(\varphi)$.

5.3 Hypotheses

As the known theoretical upper time bounds for game model checking are expressed in terms of the parameters shown in Table 1, these parameters are also the ones isolated in the inputs for the complexity verification.

Symbol	Meaning
$\|S\|$	The number of states in the game model being model checked.
$\|\mathcal{M}\|$	Size of the game model $\mathcal{M}$ being model checked (see Definition 6).
$\|\varphi\|$	The size of the formula φ (see Definition 7).
$\mathrm{ad}(\varphi)$	The alternation depth of the formula φ (see Definition 8).
t_{eg}	The execution time of a global game model check.

Table 1: Meaning of various symbols used for the experimental time complexity verification. A subset of the symbols is shared with Pauly [7].

Since any upper time-bound is ideally as strict as possible for practical purposes, we will seek to verify the more strict time-bound given in Equation 3. We can dissect the verification of the time bound into the following hypotheses, where $\forall i$: a_i, c_i and n_i are constants:

H1: $t_{eg} \leq a_1|\mathcal{M}|$ for constant $|\varphi|$, $\mathrm{ad}(\varphi) = 0$, and $|S| = |\mathcal{M}|$ $\forall |\mathcal{M}| \geq n_1$.
H2: $t_{eg} \leq a_2|\varphi|$ for constant $|\mathcal{M}|$ and $\mathrm{ad}(\varphi) = 0$ $\forall |\varphi| \geq n_2$.
H3: $t_{eg} \leq a_3|\varphi|c^{\mathrm{ad}(\varphi)}$ for constant $|\mathcal{M}|$ and constant $|S| = c_3$ $\forall\, \mathrm{ad}(\varphi) \geq n_3$.

H4: $t_{eg} \leq a_4|S|^{c_4}$ for constant $|\mathcal{M}|$, constant $|\varphi|$, and constant $\mathrm{ad}(\varphi) = c_4$ $\forall |S| \geq n_4$.
H5: $t_{eg} \leq a_5|\mathcal{M}|$ for constant $|S|$, constant $\mathrm{ad}(\varphi)$, and constant $|\varphi|$ $\forall |\mathcal{M}| \geq n_5$.
H6: $t_{eg} \leq a_6|\varphi|$ for constant $|S|$, constant $\mathrm{ad}(\varphi)$, and constant$|\mathcal{M}|$ $\forall |\varphi| \geq n_6$.
H7: $t_{eg} \leq a_7 c_7^{\mathrm{ad}(\varphi)}$ for constant $|\mathcal{M}|$, constant $|\varphi|$, and constant $|S| = c_7$ $\forall\, \mathrm{ad}(\varphi) \geq n_7$.

The hypotheses are split up into two parts: H1 up to and including H3 and H4 up to and including H7. There are some essential differences between these parts, which can be found in Table 2.

We will attempt to verify these hypotheses one by one by isolating the relevant input parameter, while fixing all the others.

H1 - H3	H4 - H7
Constant parameter values as simple (low) as possible.	Constant parameter values are higher.
Models and formulas behind constant parameter values are identical. E.g., for experiment 2: $\|S\| = 3$ means that an identical set of states with the same properties holding true in them were used for all tests in this experiment.	Models and formulas behind constant parameter values differ for each value of the isolated parameter. E.g., for experiment 4: the model and formula used for $\|S\| = 50$ are completely different from the model and formula used for $\|S\| = 60$. (Between the 5 runs for the same $\|S\|$, however, they are identical.)
Meant for initial observations to compare to the results of experiment 4 through 7.	Meant to result in strong evidence for or against the time complexity in Equation 3 holding true for the model checker.
Possibly isolate more than one parameter, making the varying parameters depend on each other.	Vary exactly one parameter, while keeping the others constant.

Table 2: Differences between the two sets of hypotheses for the experimental time complexity verification.

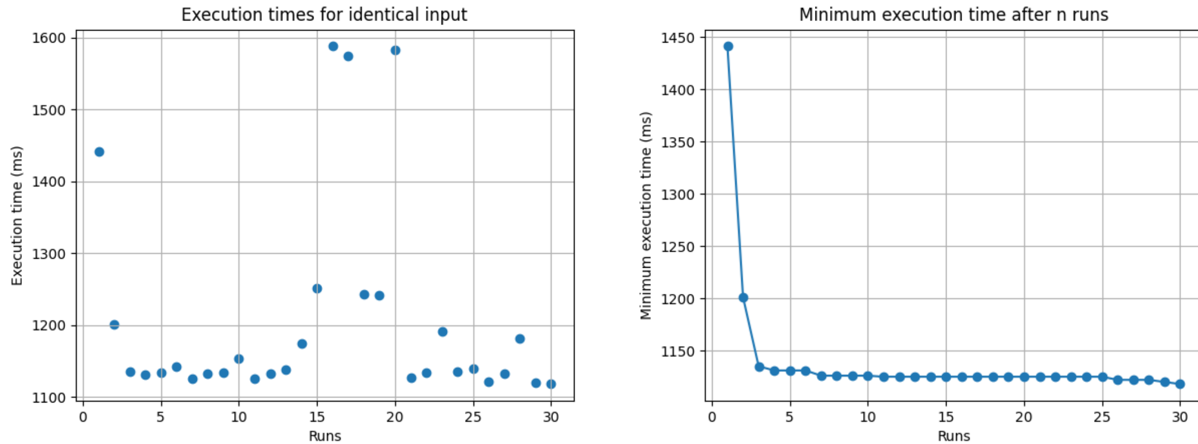
5.4 Experimental Set-Up

The experiments are set up such that each hypothesis corresponds to one experiment. The experiment that corresponds to a certain hypothesis is used exclusively to verify that hypothesis. As such, the experiments are numbered with the same index as the hypothesis they are intended to verify. Each experiment is listed in sections 5.4.2 through 5.4.8 with

1. the goal of the experiment;
2. the parameters that were kept constant and their values;
3. the parameter that was varied and its range;
4. a description of the (game model, formula) pairs that were used to obtain these parameter values for H1 through H3 (as these were randomly generated for each input for H4 through H7).

The experiments were carried out by executing the model checker with each input and measuring the time t_{eg} in milliseconds from the moment the input has been parsed until the model check result has been obtained.

This measurement was performed a number of times per (game model, formula) pair, after which the minimum of the measurements was taken as the most accurate result to alleviate the effect of background processes on the execution time. A consideration here is the number of times to run the program for each input. Figure 5.1b (data: Appendix C.1, Table 10) shows the minimum execution time for a single input ($|\mathcal{M}| = 500, |S| = 100, |\varphi| = 100, \text{ad}(\varphi) = 2$) after a number of runs. It suggests that after three to four runs, the execution time is unlikely to decrease much further. To be safe, all measurements for these experiments were performed five times before taking the minimum. These most accurate results were used to create the tables in Appendix C.1 and to generate the graphs of the results in Section 5.5.



(a) Model checker execution times measured a number of times against the same input.

(b) Model checker minimum execution times after running a number of times against the same input.

Figure 5.1: Model checker execution times and minimum times against the number of runs for identical input ($|\mathcal{M}| = 500, |S| = 100, |\varphi| = 100, \text{ad}(\varphi) = 2$).

5.4.1 Input Generation

The program used for randomly generating the inputs works by first generating the possible propositions of the model with an amount in the range $[1, \min(|S|, 500)]$ (in the description of the input generator, any range that a value is randomly chosen from has a uniform probability distribution). These propositions are given identifiers $p0, p1, ..., p(n_p - 1)$ with n_p the number of propositions.

After generating the possible propositions, a set number ($|S|$) of states is generated (with identifiers $s0, s1, ..., s(|S| - 1)$) where in each state, a random sample of propositions with a size in the range $[0, n_p]$ is true.

Next, a number $n_g = \min(|\mathcal{M}| - |S|, \lceil |\mathcal{M}|/|S| \rceil)$ of atomic games (with identifiers $g0, g1, ..., g(n_g - 1)$) and corresponding neighbourhood functions are generated, with the restriction that the sum of the cardinalities of all (not necessarily unique) neighbourhoods that can be mapped to through the neighbourhood frames is equal to $n_{s_tot} = |\mathcal{M}| - |S|$. To this end, every neighbourhood frame $nbhf_i$ is given an exact sum of neighbourhood cardinalities $n_{s_nbhf_i}$ that it must contain, with the condition that $\sum_{i=0}^{n_g - 1} n_{s_nbhf_i} = n_{s_tot}$. For each neighbourhood frame, the number of mappings it contains lies in the range $[1, n_{s_nbhf_i}]$. Each mapping map_j then has its own exact sum of neighbourhood cardinalities to contain, $n_{s_map_j}$. Accordingly, a permutation of neighbourhoods is generated for each mapping, where for each neighbourhood U_k, $|U_k|$ lies in the range $[0, \min(|S|, n_{s_map_j})]$.

Finally, the input requires a formula φ to be generated with an exact length $|\varphi|$. This is done by recursively generating smaller formulas and games which are part of φ. In the base case, $|\varphi| = 1$, an atomic game or proposition is used. We distinguish the recursive cases $|\varphi| = 2$ and $|\varphi| > 2$. When $|\varphi| = 2$, an operator that takes one parameter is randomly chosen as the formula. When $|\varphi| > 2$, any operator is randomly chosen.

To be able to generate formulas with certain alternation depths while still being randomly generated, additional recursive cases are defined that override those mentioned above. Let add(φ) be the desired remaining alternation depth in φ and let l be the desired remaining length. At any point in the recursion, if the term being generated is a game, a fixpoint operator is generated such that the alternation depth increases

with probability $2 \cdot \mathrm{add}(\varphi)/l$. If the term being generated is a formula, a diamond modality $\langle\alpha\rangle\psi$ is generated with probability $2(\mathrm{add}(\varphi)+2)/l$, giving room for the alternation depth to increase in α. This strategy does not guarantee φ to have an exact alternation depth, but it does make it highly probable.

After generating each input, an additional check was performed to ensure the correctness of the input parameters using the function `reportComplexityParams` of the `ComplexityAnalysis` module listed in Appendix D.9. If this check indicates the presence of incorrect parameters, the corresponding input is re-generated until the desired parameter values are obtained.

5.4.2 Description Experiment 1

Goal: show a linear relation between t_{eg} and $|\mathcal{M}|$ when $|S| = |\mathcal{M}|$ and $\mathrm{ad}(\varphi) = 0$. To this end, it suffices to show a linear relation between t_{eg} and $|S|$, since the expression $|S|^{\mathrm{ad}(\varphi)} \times |\mathcal{M}| \times |\varphi|$ from Equation 3 simplifies to $|S| \times |\varphi|$ under the given conditions.

Parameter	Value or range
$\|S\|$	$\|\mathcal{M}\|$
$\|\mathcal{M}\|$	[1000, 5000, 10000, ..., 150000]
$\|\varphi\|$	1
$\mathrm{ad}(\varphi)$	0

Table 3: Parameter values for experiment 1.

Let the game model for experiment 1 be $\mathcal{M}_1 = (S_1, \gamma_1, \Upsilon_1)$, then:

$$S_1 = \begin{cases} \{s_0, s_1, ..., s_{999}\} & \text{for input } 1 \\ \{s_0, s_1, ..., s_{5000(i-1)-1}\} & \text{for input } i, \forall i \in [2, 31] \end{cases}$$

$$\gamma_1 = \emptyset$$

$\Upsilon_1 = \{(p, T)\}$, where T is a set that contains a state s with probability 0.5 $\forall s \in S_1$.

$$\varphi = p$$

5.4.3 Description Experiment 2

Goal: show a linear relation between t_{eg} and $|\varphi|$ for a simple game model and no fixpoint operators present in φ.

Parameter	Value or range
$\|S\|$	3
$\|\mathcal{M}\|$	12
$\|\varphi\|$	[534, ..., 308792]
$\mathrm{ad}(\varphi)$	0

Table 4: Parameter values for experiment 2.

Let the game model for experiment 2 be $\mathcal{M}_2 = (S_2, \gamma_2, \Upsilon_2)$, then:

$$S_2 = \{s_0, s_1, s_2\}$$

γ_2 is defined by:

Game g	State s	$\langle\!\lvert g \rvert\!\rangle(s)$
g_0	s_1	$\{\{s_0, s_2\}\}$
	s_2	$\{\{s_2\}\}$
g_1	s_0	$\{\{s_0, s_2\}\}$
	s_2	$\{\{s_0, s_1, s_2\}\}$

$\Upsilon_2 = \{(p_0, \{s_0, s_2\}), (p_1, \{s_1\}), (p_2, \{s_1, s_2\})\}$

φ was randomly generated as described in Section 5.4.1, with the exception that it was generated with varying nesting depths rather than varying lengths due to a limitation of the input generator script at the time.

5.4.4 Description Experiment 3

Goal: show an approximately exponential relation between t_{eg} and $\text{ad}(\varphi)$ when $|\varphi| = \text{ad}(\varphi)+3$ (this condition is alleviated in experiment 7, Section 5.4.8).

Parameter	Value or range
$\lvert S \rvert$	3
$\lvert \mathcal{M} \rvert$	12
$\lvert \varphi \rvert$	$\text{ad}(\varphi) + 3$
$\text{ad}(\varphi)$	[6, 7, ..., 29]

Table 5: Parameter values for experiment 3.

The game model $\mathcal{M}_3$ for experiment 3 is identical to $\mathcal{M}_2$.

$\varphi = \langle((((g_1^{\times})^*)...)^*\rangle p_2$, where the number of fixpoint operators is equal to $\text{ad}(\varphi)$ and the outermost and innermost fixpoint operators vary.

5.4.5 Description Experiment 4

Goal: show a quadratic relation between t_{eg} and $|S|$ when $\text{ad}(\varphi) = 2$.

Parameter	Value(s)
$\lvert S \rvert$	[50, 60, ..., 490]
$\lvert \mathcal{M} \rvert$	500
$\lvert \varphi \rvert$	100
$\text{ad}(\varphi)$	2

Table 6: Parameter values for experiment 4.

5.4.6 Description Experiment 5

Goal: show a linear relation between t_{eg} and $\mathcal{M}$.

Parameter	Value(s)
$\|S\|$	25
$\|\mathcal{M}\|$	[5004, ..., 200146] with steps of ∼5000
$\|\varphi\|$	200
ad(φ)	2

Table 7: Parameter values for experiment 5.

5.4.7 Description Experiment 6

Goal: show a linear relation between t_{eg} and $|\varphi|$.

Parameter	Value(s)
$\|S\|$	[100, 200, ..., 5000]
$\|\mathcal{M}\|$	400
$\|\varphi\|$	100
ad(φ)	2

Table 8: Parameter values for experiment 6.

5.4.8 Description Experiment 7

Goal: show an exponential relation between t_{eg} and ad(φ).

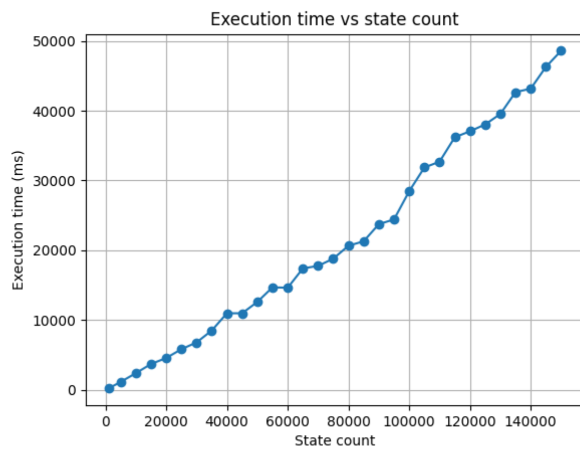
Parameter	Value(s)
$\|S\|$	5
$\|\mathcal{M}\|$	50
$\|\varphi\|$	500
ad(φ)	[0, 1, ..., 20]

Table 9: Parameter values for experiment 7.

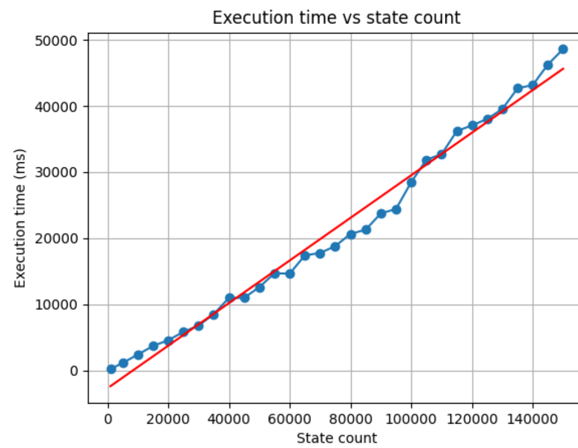
5.5 Results

5.5.1 Results Experiment 1

The result data for experiment 1 are shown in Figure 5.2 and Table 11. The graphs seem to suggest that t_{eg} may be linearly distributed in terms of the state count, as it stays close to the regression line. However, Figure 5.2b shows that the data points form a convex curve under the linear regression line, indicating that the relation between t_{eg} and $|S|$ may actually be greater than linear.



(a) Model checker execution times against varying state counts for experiment 1.

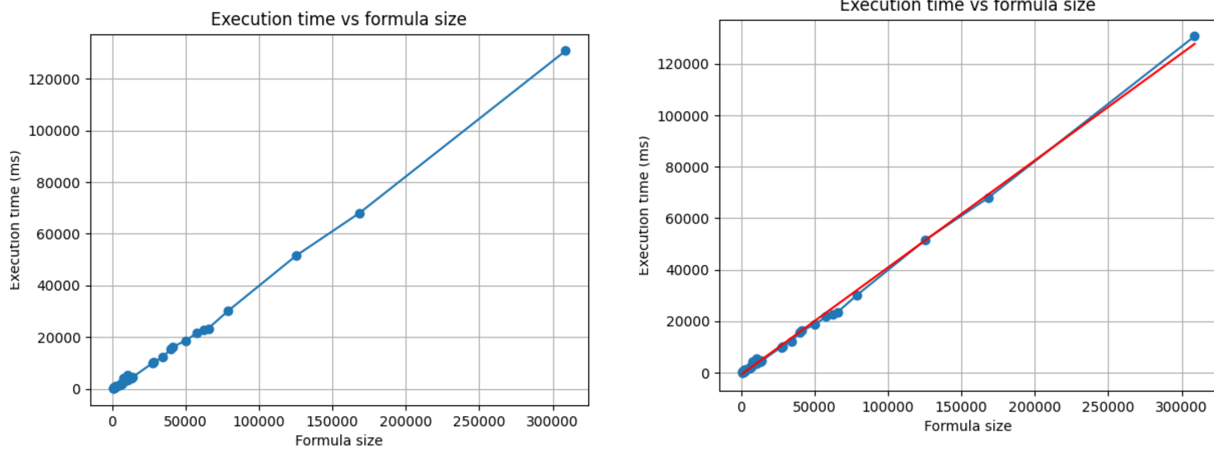


(b) Model checker execution times against varying state counts with a least-squares regression line ($y = 0.3220x - 2685.9835$) for experiment 1.

Figure 5.2: Model checker execution times measured against state count for experiment 1.

5.5.2 Results Experiment 2

The result data for experiment 2 are shown in Figure 5.3 and Table 12. The graphs show that t_{eg} seems to be linearly related to $|\varphi|$, meaning that H2 is likely correct.



(a) Model checker execution times against varying formula sizes for experiment 2.

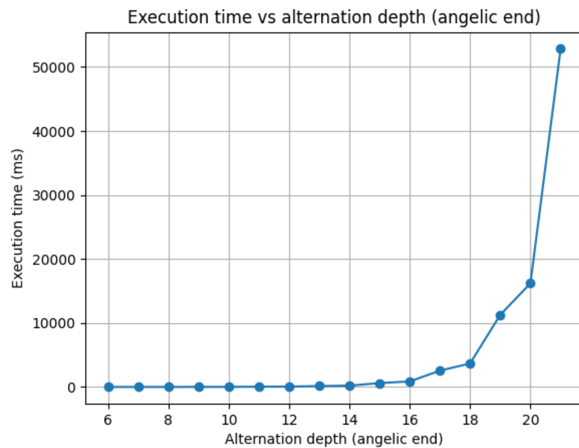
(b) Model checker execution times against varying formula sizes with a least-squares regression line ($y = 0.4161x - 737.6438$) for experiment 2.

Figure 5.3: Model checker execution times measured against formula size for experiment 2.

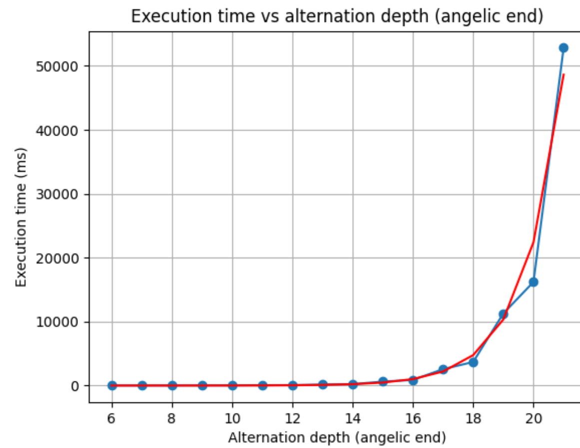
5.5.3 Results Experiment 3

The result data for experiment 3 are shown in Figure 5.4 and 5.5 and Table 13 and 14 respectively.

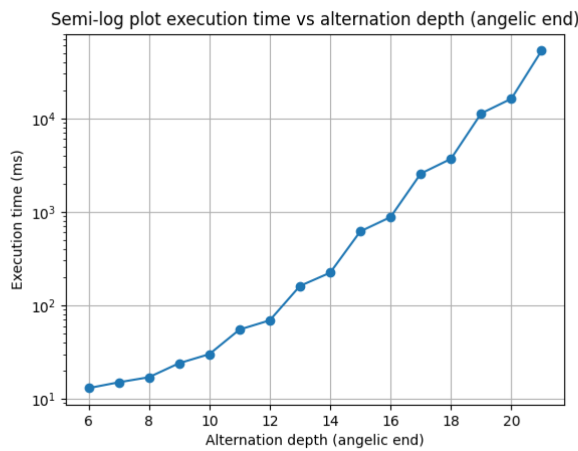
For this experiment, initial trials resulted in extremely fluctuating graphs. This fluctuation was largely eliminated by separating the input sets into one with formulas in which the alternation depth ends in an angelic iteration (Figure 5.4, Table 13) and one where it ends in a demonic iteration (Figure 5.5, Table 14). In the resulting semi-log plots, the graphs seem to neatly follow a line as the alternation depth increases, indicating a possibility that t_{eg} is exponentially related to $\mathrm{ad}(\varphi)$. Interestingly, the tests with an angelic ending to the alternation increase in run time much faster than those ending with a demonic iteration. Whether this observation holds for more complicated models and formulas is tested in experiment 7, Section 5.4.8 and 5.5.7. Finally, a consistent oscillating pattern can be observed in all of the graphs of 5.4 and 5.5. Perhaps the run time is slightly affected by whether the alternation depth starts and ends with the same type of iteration as well.



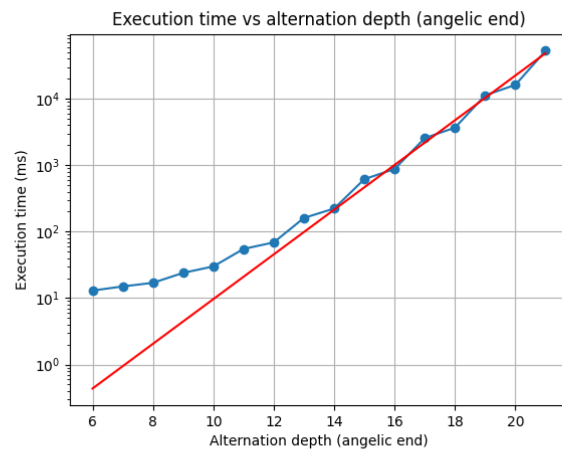
(a) Model checker execution times against varying formula alternation depths, where the alternations end in an angelic iteration of an atomic game, for experiment 3.



(b) Model checker execution times against varying formula alternation depths, where the alternations end in an angelic iteration of an atomic game (an angelic iteration is the innermost fixpoint operator of the alternation). The red curve is a least-squares regression on the natural logarithm of the execution time ($\log_{10}(y) = 0.7748x - 5.4798$), weighted with the square root of the execution time to eliminate bias against high values, then taken as a power of e to invert the logarithm.

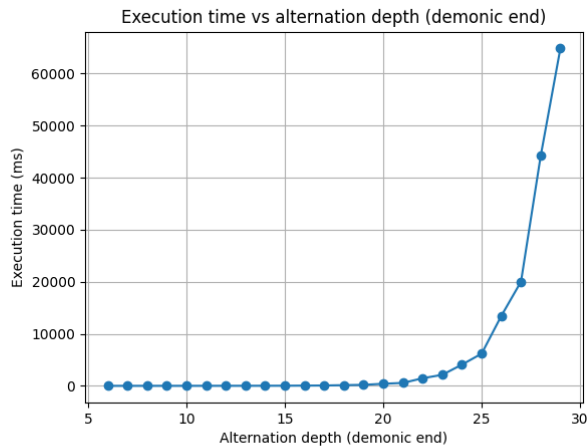


(c) Semi-log plot of model checker execution times against varying formula alternation depths, where the alternations end in an angelic iteration of an atomic game, for experiment 3.

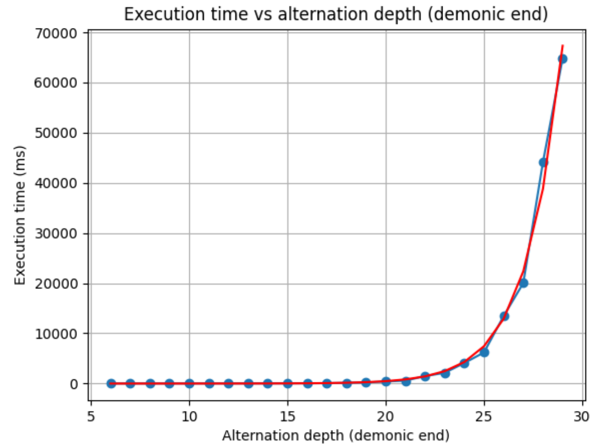


(d) Semi-log plot of model checker execution times against varying formula alternation depths, where the alternations end in a angelic iteration of an atomic game (an angelic iteration is the innermost fixpoint operator of the alternation), with a least-squares regression line ($\log_{10}(y) = 0.7748x - 5.4798$) for experiment 3.

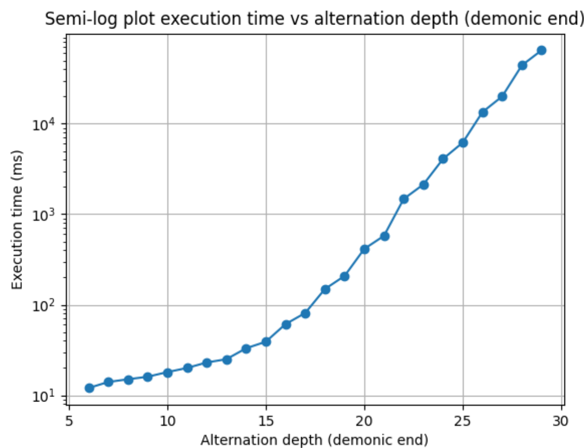
Figure 5.4: Model checker execution times measured against alternation depth with an angelic iteration as the innermost fixpoint operator for experiment 3.



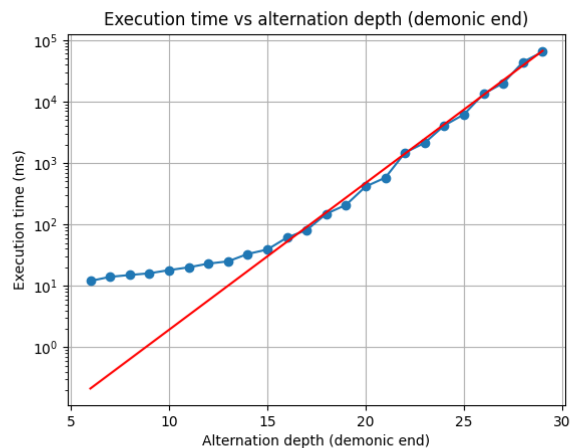
(a) Model checker execution times against varying formula alternation depths, where the alternations end in a demonic iteration of an atomic game, for experiment 3.



(b) Model checker execution times against varying formula alternation depths, where the alternations end in a demonic iteration of an atomic game. The red curve is a least-squares regression on the natural logarithm of the execution time ($\log_{10}(y) = 0.5507x - 4.8549$), weighted with the square root of the execution time to eliminate bias against high values, then taken as a power of e to invert the logarithm.



(c) Semi-log plot of model checker execution times against varying formula alternation depths, where the alternations end in a demonic iteration of an atomic game, for experiment 3.



(d) Semi-log plot of model checker execution times against varying formula alternation depths, where the alternations end in a demonic iteration of an atomic game, with a least-squares regression line ($\log_{10}(y) = 0.5507x - 4.8549$) for experiment 3.

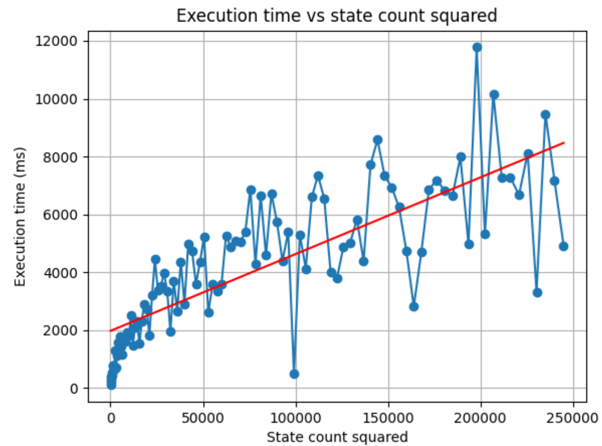
Figure 5.5: Model checker execution times measured against alternation depth with a demonic iteration as the innermost fixpoint operator for experiment 3.

5.5.4 Results Experiment 4

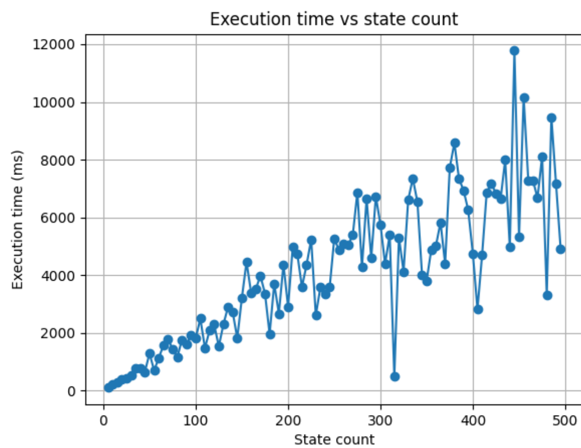
The result data for experiment 4 are shown in Figure 5.6 and Table 15.

Since $\text{ad}(\varphi) = 2$ for experiment 4, it is to be expected that $t_{eg} = a|S|^2 + b$ here. I.e., t_{eg} is expected to form a parabola when plotted against $|S|$, and to form a straight line when plotted against $|S|^2$. However, Figure 5.6b shows that t_{eg} does not seem to be quadratic in $|S|$. Additionally, 5.6d shows that t_{eg} actually forms a (jagged) straight line when graphed against $|S|$, suggesting a linear relation. The jaggedness of the line also indicates that some variables outside the scope of the experiment may be affecting t_{eg}, as it increases with $|S|$. Random fluctuations caused by background processes would be independent of $|S|$.

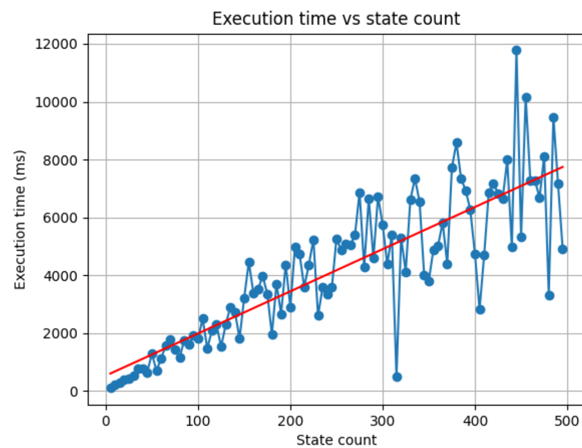
(a) Model checker execution times against varying squared state counts for experiment 4.



(b) Model checker execution times against varying squared state counts with a least-squares regression line ($y = 0.0265x + 1969.8433$) for experiment 4.



(c) Model checker execution times against varying state counts for experiment 4.

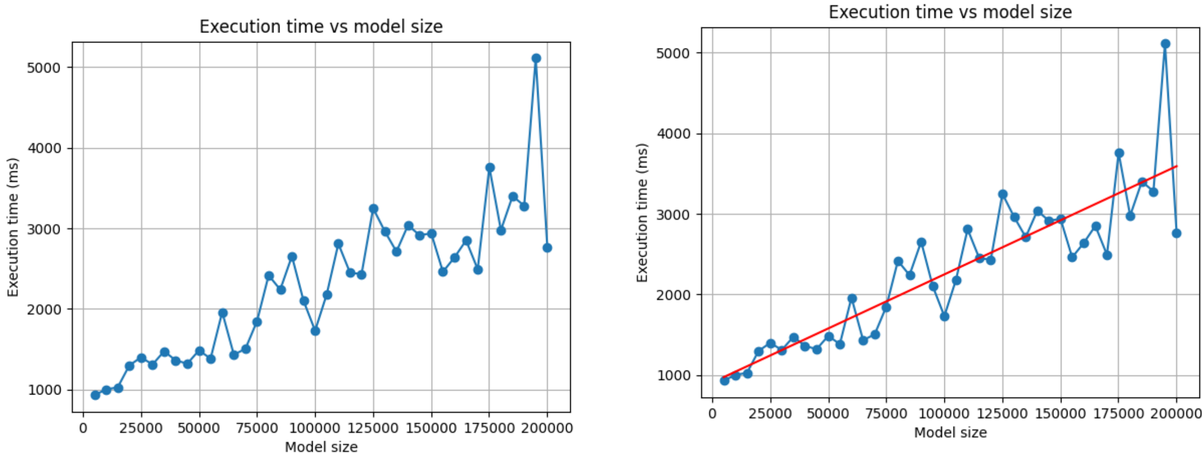


(d) Model checker execution times against varying state counts with a least-squares regression line ($y = 14.5778x + 525.0625$) for experiment 4.

Figure 5.6: Model checker execution times measured against state count for experiment 4.

5.5.5 Results Experiment 5

The result data for experiment 5 are shown in Figure 5.7 and table and 16. Here, t_{eg} seems to be linearly related to $|\mathcal{M}|$, but the data points form a very jagged line, becoming more jagged as $|\mathcal{M}|$ increases. Since background processes that are unrelated to the model checker would randomly increase the execution time independently of $|\mathcal{M}|$, the results suggest that there may be factors influencing t_{eg} here that fall outside the scope of the experiment.



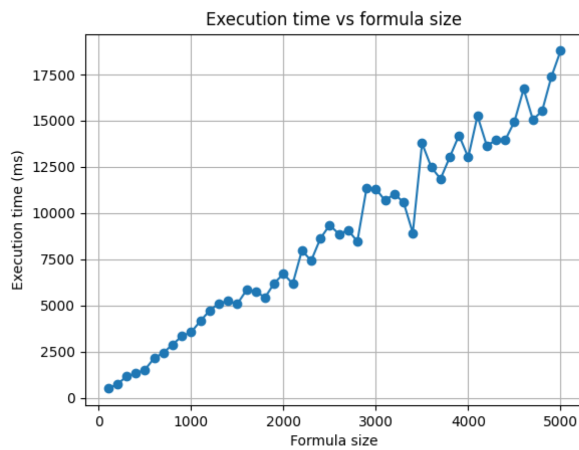
(a) Model checker execution times against varying model sizes for experiment 5.

(b) Model checker execution times against varying model sizes with a least-squares regression line ($y = 0.0134x + 908.0869$) for experiment 5.

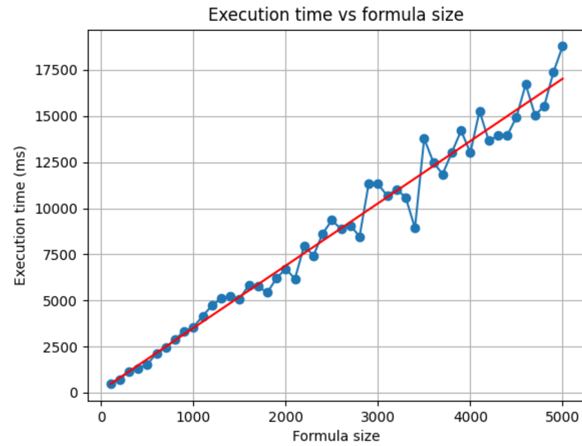
Figure 5.7: Model checker execution times measured against model size for experiment 5.

5.5.6 Results Experiment 6

The result data for experiment 6 are shown in Figure 5.8 and Table 17. As also suggested in the results of experiment 2, t_{eg} again seems to be linearly related to $|\varphi|$, since the data form quite a neat line along the linear regression line. Furthermore, fluctuations in the execution time again become greater as the free variable ($|\varphi|$) grows, hinting at the existence of unknown variables affecting t_{eg}.



(a) Model checker execution times against varying formula sizes for experiment 6.



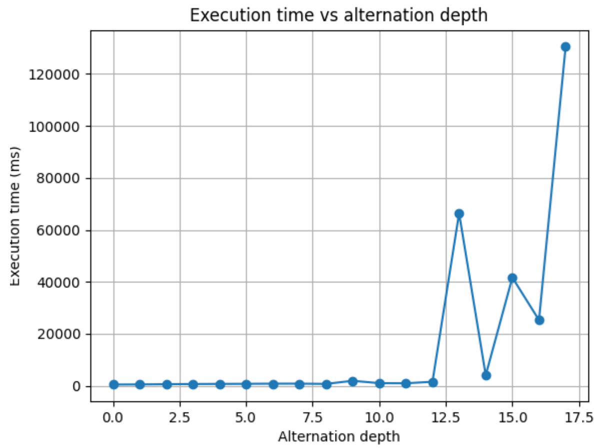
(b) Model checker execution times against varying formula sizes with a least-squares regression line ($y = 3.3807x + 122.3469$) for experiment 6.

Figure 5.8: Model checker execution times measured against formula size for experiment 6.

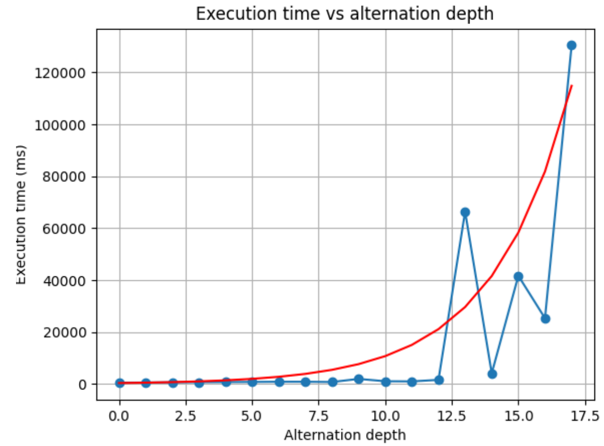
5.5.7 Results Experiment 7

The result data for experiment 7 are shown in Figure 5.9, 5.10, and 5.11 and Table 18, 19, and 20 respectively.

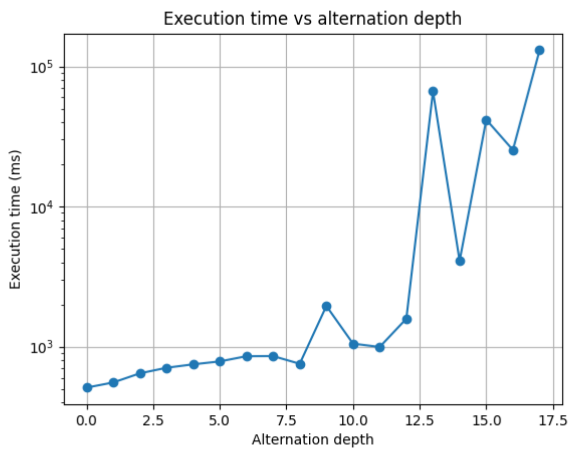
Having seen the results of experiment 3, we expect to see a considerable difference here between the effect of alternations that end in an angelic iteration and alternations that end in a demonic iteration. What we observe instead is rather unpredictable execution times regardless of angelic end or demonic end. In each of the 3 test sets (angelic end, demonic end, either end), the data points form a relatively neat line up to $\text{ad}(\varphi) = 12$, after which the trend becomes completely inconclusive.



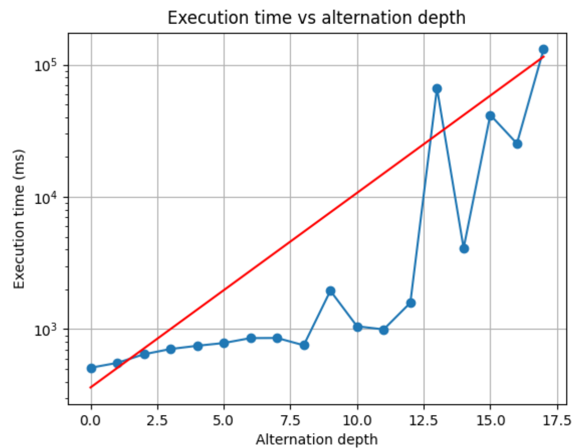
(a) Model checker execution times against varying alternation depths for experiment 7.



(b) Model checker execution times against varying alternation depths. The red curve is a least-squares regression on the natural logarithm of the execution time ($\log_{10}(y) = 0.3387x + 5.8936$), weighted with the square root of the execution time to eliminate bias against high values, then taken as a power of e to invert the logarithm.

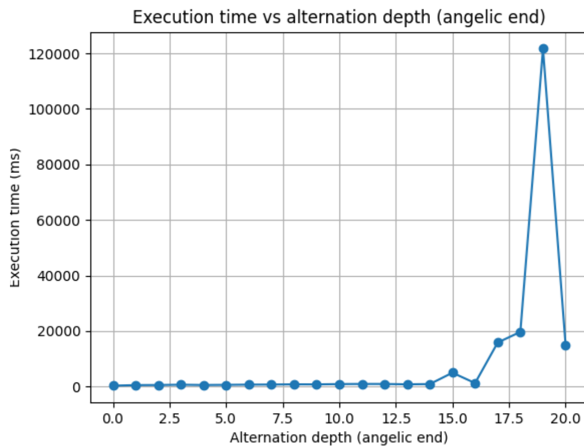


(c) Semi-log plot of model checker execution times against varying alternation depths for experiment 7.

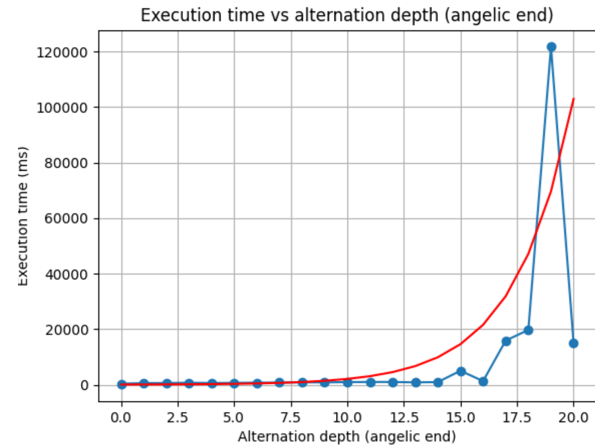


(d) Semi-log plot of model checker execution times against varying alternation depths with a least-squares regression line ($\log_{10}(y) = 0.3387x + 5.8936$) for experiment 7.

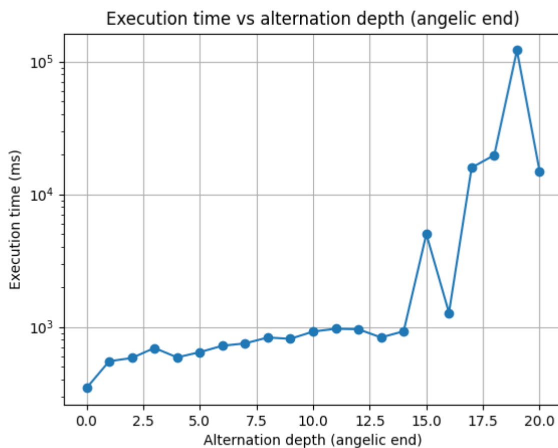
Figure 5.9: Model checker execution times measured against alternation depth for experiment 7.



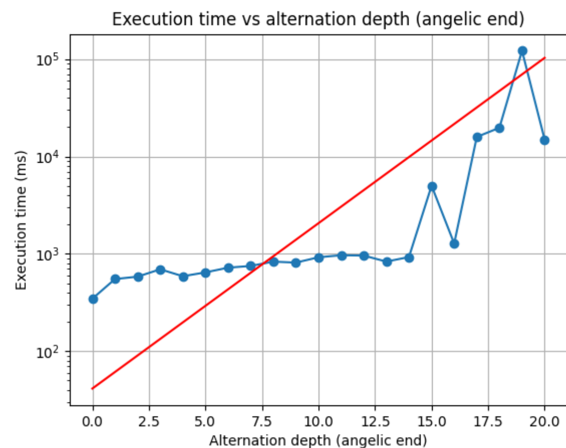
(a) Model checker execution times against varying alternation depths, where the alternations end in an angelic iteration, for experiment 7.



(b) Model checker execution times against varying alternation depths, where the alternations end in an angelic iteration. The red curve is a least-squares regression on the natural logarithm of the execution time ($\log_{10}(y) = 0.3909x + 3.7231$), weighted with the square root of the execution time to eliminate bias against high values, then taken as a power of e to invert the logarithm.

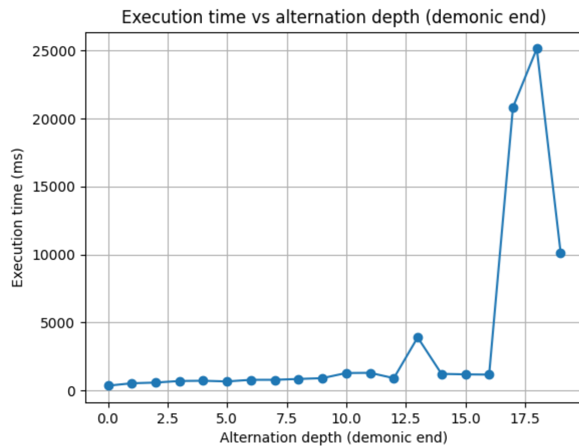


(c) Semi-log plot of model checker execution times against varying alternation depths, where the alternations end in an angelic iteration, for experiment 7.

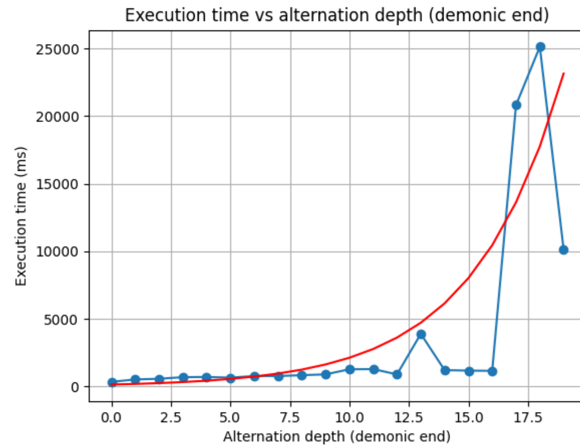


(d) Semi-log plot of model checker execution times against varying alternation depths, where the alternations end in an angelic iteration, with a least-squares regression line ($\log_{10}(y) = 0.3909x + 3.7231$) for experiment 7.

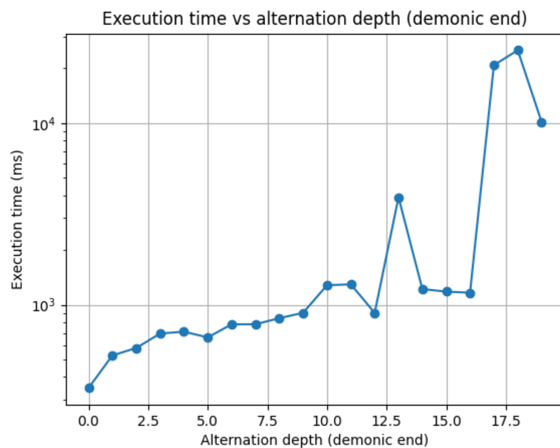
Figure 5.10: Model checker execution times measured against alternation depth with an angelic iteration as the innermost fixpoint operator for experiment 7.



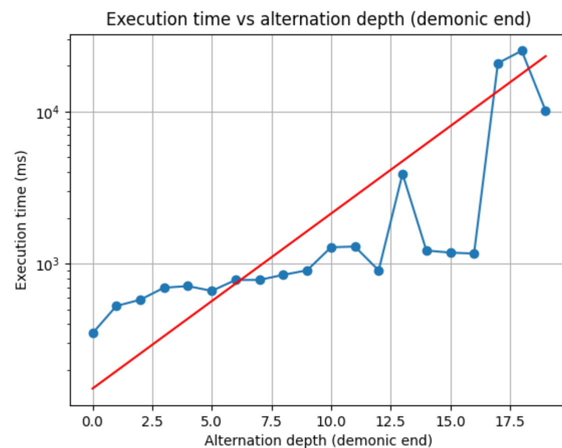
(a) Model checker execution times against varying alternation depths, where the alternations end in a demonic iteration, for experiment 7.



(b) Model checker execution times against varying alternation depths, where the alternations end in a demonic iteration. The red curve is a least-squares regression on the natural logarithm of the execution time ($\log_{10}(y) = 0.2650x + 5.0146$), weighted with the square root of the execution time to eliminate bias against high values, then taken as a power of e to invert the logarithm.



(c) Semi-log plot of model checker execution times against varying alternation depths, where the alternations end in a demonic iteration, for experiment 7.



(d) Semi-log plot of model checker execution times against varying alternation depths, where the alternations end in a demonic iteration, with a least-squares regression line ($\log_{10}(y) = 0.2650x + 5.0146$) for experiment 7.

Figure 5.11: Model checker execution times measured against alternation depth with a demonic iteration as the innermost fixpoint operator for experiment 7.

5.6 Conclusion

Strong evidence was found for hypotheses H1, H2, and H3, under the premises that the models and formulas used are rather rudimentary and identical for each test case within each experiment. Given that these are strong conditions, no conclusions can be drawn about the time complexity of the model checker or the theoretical time complexity in Equation 3 by only considering these experiments. They do, however, provide additional insight in the validity of hypotheses 4 through 7. For example, experiment 1 demonstrates a probable linear relation between $|S|$ and t_{eg}, as well as between $|\mathcal{M}|$ and t_{eg}, which is expected to turn non-linear for alternation depths greater than 1, but experiment 4 (especially Figure 5.6b) suggests that this may not hold, at least for $\text{ad}(\varphi) = 2$. Experiment 5 strengthens the observations from experiment 1, as a linear relation between $|\mathcal{M}|$ and t_{eg} seems to hold here without the premise of simple and constant models and formulas as well. Additionally, experiment 5 demonstrates that there may be variables influencing t_{eg} which are outside the scope of the current experiments, as fluctuations in the execution time increase with $|\mathcal{M}|$. Experiment 6 reinforces this possibility, as here, the fluctuations in t_{eg} also increase with $|\varphi|$. Furthermore, it shows a rather close fit of the data points to the regression line in Figure 5.8b, providing strong evidence towards the validity of H6. From experiment 3, alternation depths ending in angelic iterations seemed to have a much longer run time than those ending in a demonic iteration. Experiment 7, however, shows no such observation to be made in the more general complex case. Unfortunately, this is all that experiment 7 demonstrates, as t_{eg} becomes increasingly unpredictable with increasing alternation depths. Additionally, the run time of the model checker increases so rapidly that obtaining a large number of data points for experiment 7 was infeasible on the utilized hardware.

Out of the hypotheses with potential to make strong claims regarding the theoretical time complexity of Equation 3 in relation to the model checker (H4 up to and including H7), only H6 was verified with reasonable certainty. Experiment 4 provided evidence towards a possibly tighter time bound in regards to $|S|$ than set forth in H4. H5 and especially H7 require more extensive analysis to make any significant claims concerning the time complexity of the model checker or the practical validity of Equation 3.

6 Conclusion

6.1 General Conclusions

In this work, we set out to answer two questions: 1) “How can the model checking problem for game logic be programmatically converted to a parity game representation for an existing parity game solver?” and 2) “How can the output of this solver be converted back to a model check result for the game model?”. An answer to both of these questions has been given through the model checker that was implemented, which can be found in Appendix D and installed by the steps given in Section 3.4.

Of greater interest than the singular answers that were given here are the various ways in which the answers could be different or improved. Considerations were made regarding the programming language (Section 3.3.1), the time complexity (Section 3.3.5, 5), the choice of parity game solver (Section 3.3.2), and the input (Section 3.3.3) and output (Section 3.3.6) format. Suboptimalities in each of these are discussed in Section 6.2.

Considering the performance of the application, an input with a model size of 2000 and a formula size of 4000 can be processed in just over 15 minutes on a machine with an Intel Core i7-9750H processor at 2.6 GHz with 4 GB of memory allocated, running on Ubuntu 18.04. A more nuanced analysis of the time complexity can be found in Section 5. Whether this performance is of any use is difficult to assess at this time, as concrete practical applications for a game logic model checker are scarce at best. Regardless, there is promising room for improvement, which is considered in Section 6.2.

In spite of all the enhancements that the model checker could still undergo, the tests that were carried out and the implemented functionality show that the project has been a success at its core. With the model checker that has been implemented, the creation of game models that adhere to predefined conditions has

been made less prone to human error. Ideally, the tool will spark interest in other researchers to create superior versions and one day be utilized to ensure the correctness of an actual protocol backed by a game model.

6.2 Future Work

There is a lot of potential for future work regarding both the model checker itself and the time complexity analysis, each of which are discussed here separately. In addition to these, the issue of finding concrete instances of practical applications for a model checker for game logic persists, a search for which is a possible piece of future work in itself.

6.2.1 Program Modifications

As mentioned in Section 3.3.5, there is still room for optimization in the model checker. For one, the time complexity of the parity function procedure can be greatly improved upon by, e.g., performing a bottom-up recursion over the formula once and saving the calculated priorities in its nodes (by changing the `AFormula` data structure, Appendix D.5) or in a `map`. Using this method, the parity function could be executed in constant time, or even be removed from the program in favour of a simple look-up every time it would otherwise be used.

Furthermore, the strategy that was chosen for the execution of a global model check, solving the generated parity game globally and reporting back with the subset of the positions that correspond to the correct (state, formula) pairs for the game model, could be suboptimal for certain inputs. For further optimization, tests could be performed with game models and formulas with different parameters to see whether the current strategy is optimal for them or one with several sequential local solves of the parity game.

Another performance increase could be achieved by including more, varying parity game solvers in the program. In particular, the implementation of Zielonka's algorithm [28] in the more modern collection of parity game solvers, called Oink, has been shown to outperform the PGSolver implementation [12]. In the case of such a one-to-one improvement, the model checker could be modified to use the Oink implementation instead. Additionally, the Oink solvers could be analyzed to see if they contain a solver that is more fitting to be the default for the model checker than the big step algorithm [27]. In general, including more parity game solvers leads to more flexibility in the types of game models and formulas which can be model checked efficiently by a well-informed user.

In addition to performance increases, there are additions to be made to the program to fulfill the remaining parts of the requirements. First, the functionality for R-3.3 can be extended to include the moves of player 2 as well. The PGSolver output does not contain any transitions from a neighbourhood to a particular state, leaving out half of the play in the model check results. To accommodate for moves for player 2 as well, the model checking problem itself could be inspected, as opposed to the corresponding evaluation game, to find out how the diamond modalities unfold for particular inputs and give a more elaborate description of the play this way.

Finally, an input format could be added to the model checker that is more representative of game logic syntax for fomulas (R-3.2). To this end, inspiration can be taken from Worthington's proof transformation tool for game logic [29], also written in Rascal.

6.2.2 Time Complexity Evaluation

When it comes to the time complexity evaluation, we can divide the future work up into two categories: what requires more investigation and how this investigation can be sustained. We start by considering the former.

As can be observed by the results of experiment 4, there appears to exist a time bound for t_{eg} in terms of the alternation depth that is sharper than the one described in Equation 3. This relation could be

investigated further by varying the state count for more alternation depths (as experiment 4 limited itself to $\mathrm{ad}(\varphi) = 2$ to accommodate for higher state counts), and observing whether this results in t_{eg} being quadratic, cubic, quartic, ... in $|S|$.

Due to the large fluctuations that were observed in the results of experiment 5, 6, and 7, conclusions regarding the time complexity of the model checker could only be hinted at, rather than shown with strong evidence. To diminish the amount of fluctuation in these results, tests could be performed in a similar manner, but averaging t_{eg} for different randomly generated inputs with the same parameter values and plotting this average instead of a singular measurement.

The observation of fluctuations becoming greater with larger values of the running variable introduces the possibility that parameters outside the scope of Equation 3 may be influencing t_{eg}. After implementing the optimizations to the model checker mentioned in Section 6.2.1, the hypotheses could be re-evaluated to see if the fluctuations persist. If so, the possibility of other variables being at play could be investigated by using different ways of generating random inputs, by closely inspecting the program itself, or by gaining more results through the use of larger test sets, a more powerful machine, more computation time, or the use of specialized parity game solvers for each experiment.

References

[1] E. W. Dijkstra, “The humble programmer,” Communications of the ACM, vol. 15, no. 10, p. 859–866, 1972.

[2] C. Stirling and D. Walker, “Local model checking in the modal mu-calculus,” in TAPSOFT ’89 (J. Díaz and F. Orejas, eds.), (Berlin, Heidelberg), pp. 369–383, Springer Berlin Heidelberg, 1989.

[3] M. Y. Vardi, “An automata-theoretic approach to linear temporal logic,” in Logics for Concurrency, p. 238–266, Springer Berlin Heidelberg, 1996.

[4] T. Hafer and W. Thomas, “Computation tree logic ctl* and path quantifiers in the monadic theory of the binary tree,” in Automata, Languages and Programming, p. 269–279, Springer, 1987.

[5] L. Lamport, “The temporal logic of actions,” ACM Trans. Program. Lang. Syst., vol. 16, no. 3, p. 872–923, 1994.

[6] R. Parikh, “Propositional game logic,” in 24th Annual Symposium on Foundations of Computer Science (sfcs 1983), IEEE, 1983.

[7] M. Pauly, Logic for Social Software. PhD thesis, University of Amsterdam, 2001.

[8] D. Berwanger, “Game logic is strong enough for parity games,” Studia Logica, vol. 75, no. 2, p. 205–219, 2003.

[9] S. Enqvist, H. H. Hansen, C. Kupke, J. Marti, and Y. Venema, “Completeness for game logic,” in 2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), IEEE, 2019.

[10] H. H. Hansen, C. Kupke, J. Marti, and Y. Venema, “Parity games and automata for game logic,” in Dynamic Logic. New Trends and Applications (A. Madeira and M. Benevides, eds.), (Cham), pp. 115–132, Springer International Publishing, 2018.

[11] O. Friedmann and M. Lange, “The pgsolver collection of parity game solvers.” *Tue.nl* [Online] Available: `https://www.win.tue.nl/~timw/downloads/amc2014/pgsolver.pdf`, 2014. [Accessed: 03-May-2022].

[12] T. van Dijk, “Oink: An implementation and evaluation of modern parity game solvers,” in Tools and Algorithms for the Construction and Analysis of Systems (D. Beyer and M. Huisman, eds.), (Cham), pp. 291–308, Springer International Publishing, 2018.

[13] J. Fearnley, “Efficient parallel strategy improvement for parity games,” in Computer Aided Verification (R. Majumdar and V. Kunčak, eds.), (Cham), pp. 137–154, Springer International Publishing, 2017.

[14] M. J. Fischer and R. E. Ladner, “Propositional dynamic logic of regular programs,” Journal of computer and system sciences, vol. 18, no. 2, p. 194–211, 1979.

[15] H. H. Hansen, “Monotonic modal logic,” Master’s thesis, University of Amsterdam, 2003.

[16] M. Pauly and R. Parikh, “Game logic - an overview,” Studia Logica, vol. 75, no. 2, p. 165–182, 2003.

[17] E. Grädel, W. Thomas, and T. Wilke, “Parity games,” in Automata, Logics, and Infinite Games, (Berlin, Heidelberg), p. 94–131, Springer, 2002.

[18] R. Cleaveland and B. Steffen, “A linear-time model-checking algorithm for the alternation-free modal mu-calculus,” Formal methods in system design, vol. 2, no. 2, p. 121–147, 1993.

[19] O. V. Sokolsky and S. A. Smolka, “Incremental model checking in the modal mu-calculus,” in Computer Aided Verification, p. 351–363, Springer Berlin Heidelberg, 1994.

[20] R. Cleaveland, M. Klein, and B. Steffen, “Faster model checking for the modal mu-calculus,” in Computer Aided Verification, p. 410–422, Springer Berlin Heidelberg, 1993.

[21] J. Bradfield and I. Walukiewicz, “The mu-calculus and model checking,” in Handbook of Model Checking, p. 871–919, Springer International Publishing, 2018.

[22] Y. Okawa and T. Yoneda, “Symbolic computation tree logic model checking of time petri nets,” Electronics and Communications in Japan. Part 3, Fundamental Electronic Science, vol. 80, no. 4, p. 11–20, 1997.

[23] P. Baltazar, R. Chadha, and P. Mateus, “Quantum computation tree logic — model checking and complete calculus,” International journal of quantum information, vol. 6, no. 2, p. 219–236, 2008.

[24] D. Kernberger and M. Lange, “Model checking for the full hybrid computation tree logic,” in 2016 23rd International Symposium on Temporal Representation and Reasoning (TIME), pp. 31–40, IEEE, 2016.

[25] Y. Li, Y. Li, and Z. Ma, “Computation tree logic model checking based on possibility measures,” Fuzzy Sets and Systems, vol. 262, pp. 44–59, 2015.

[26] Y. Li, L. Lei, and S. Li, “Computation tree logic model checking based on multi-valued possibility measures,” Information sciences, vol. 485, p. 87–113, 2019.

[27] S. Schewe, “Solving parity games in big steps,” in FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science (V. Arvind and S. Prasad, eds.), (Berlin, Heidelberg), pp. 449–460, Springer Berlin Heidelberg, 2007.

[28] W. Zielonka, “Infinite games on finitely coloured graphs with applications to automata on infinite trees,” Theoretical Computer Science, vol. 200, no. 1, pp. 135–183, 1998.

[29] C. Worthington, “Proof transformations for game logic.,” Bachelor’s thesis, University of Groningen, 2021.

7 Appendix

A Proofs

Theorem 1. *For any game model $\mathcal{M} = (S, \gamma, \Upsilon)$: $\mathcal{M}, s \models \varphi$ iff $\text{nmc}(\mathcal{M}), s \models \varphi$, $\forall s \in S, \forall \varphi \in \mathcal{F}$.*

Proof. Since $\text{nmc}(\mathcal{M})$ is identical to $\mathcal{M}$ up to its collection of neighbourhood frames, the theorem is trivially true for any formula that does not concern a state transition (anything but a diamond modality). As such, it suffices to show that *"for any game model $\mathcal{M} = (S, \gamma, \Upsilon)$: $\mathcal{M}, s \models \langle\alpha\rangle\psi$ iff $\text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi$, $\forall s \in S, \forall \alpha \in \mathcal{G}, \forall \psi \in \mathcal{F}$, where β is the game in $\text{nmc}(\mathcal{M})$ which was constructed from α"* (1). By β being constructed from α, we mean that all the subgames of β were recursively constructed from their corresponding subgames of α, down to the level of atomic games. Due to the semantics in Definition 4, we can then be sure that for any neighbourhood $U \in \langle\!\langle\alpha\rangle\!\rangle(s)$, either $U \in \langle\!\langle\beta\rangle\!\rangle(s)$ or $U \supseteq U' \in \langle\!\langle\beta\rangle\!\rangle(s)$ for all $s \in S$.

First, we show that $\text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi \Rightarrow \mathcal{M}, s \models \langle\alpha\rangle\psi$ (2). To this end, consider the situation in which $\text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi$, then Angel has a strategy in β to ensure an outcome state in which ψ holds starting from s. In other words, Angel chooses a neighbourhood $U \in \langle\!\langle\beta\rangle\!\rangle(s)$ such that $\text{nmc}(\mathcal{M}), t \models \psi\ \forall t \in U$. Since $\langle\!\langle\alpha\rangle\!\rangle(s)$ is a superset of $\langle\!\langle\beta\rangle\!\rangle(s)$, we know that $U \in \langle\!\langle\alpha\rangle\!\rangle(s)$. Thus, Angel can choose the neighbourhood U to transition to when playing α in the state s, ensuring an outcome state in which ψ holds. Therefore, $\text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi \Rightarrow \mathcal{M}, s \models \langle\alpha\rangle\psi$.

Next, we show that $\mathcal{M}, s \models \langle\alpha\rangle\psi \Rightarrow \text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi$ (3). Consider the situation in which $\mathcal{M}, s \models \langle\alpha\rangle\psi$, then Angel has a strategy in α to ensure an outcome state in which ψ holds starting from s. In other words, Angel chooses a neighbourhood $U \in \langle\!\langle\alpha\rangle\!\rangle(s)$ such that $\mathcal{M}, t \models \psi\ \forall t \in U$. Since $\langle\!\langle\alpha\rangle\!\rangle(s) = \langle\!\langle\beta\rangle\!\rangle(s) \cup (\langle\!\langle\alpha\rangle\!\rangle(s) \setminus \langle\!\langle\beta\rangle\!\rangle(s))$, and $\langle\!\langle\alpha\rangle\!\rangle(s) \setminus \langle\!\langle\beta\rangle\!\rangle(s)$ only contains neighbourhoods of which subsets exist in $\langle\!\langle\beta\rangle\!\rangle(s)$, we know that $\langle\!\langle\beta\rangle\!\rangle(s)$ contains some set $U' \subseteq U$. Furthermore, since $\mathcal{M}, t \models \psi\ \forall t \in U$, necessarily $\mathcal{M}, t \models \psi\ \forall t \in U'$. Thus, Angel can choose the neighbourhood U' to transition to when playing β in the state s, ensuring an outcome state in which ψ holds. Therefore, $\mathcal{M}, s \models \langle\alpha\rangle\psi \Rightarrow \text{nmc}(\mathcal{M}), s \models \langle\beta\rangle\psi$.

Combining (2) and (3), we conclude that the bi-implication (1) holds, proving the theorem.

B Program Testing

B.1 Small Inputs

```
model
state0 prop0,
state1 prop1 prop2 prop3,
state2 prop2,
state3 prop3 prop0,
state4 prop1 prop4 prop6 prop7,
state5 prop5 prop4 prop0 prop9,
state6 ,
state7 prop0 prop7,
state8 prop7;
end model
prop0
```

Listing 2: Small input for the test case concerning the atomic proposition.

```
The formula
  "prop("prop0")"
is true in states
  {"state3","state5","state7","state0"}
and false in states
  {"state1","state2","state4","state6","state8"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
  state4 -> stay
  state5 -> stay
  state6 -> stay
  state7 -> stay
  state8 -> stay
  state0 -> stay
```

Listing 3: Output of the test case concerning the atomic proposition.

```
model
state1 prop1,
state2;
end model
not(prop1)
```

Listing 4: Small input for the test case concerning the negation.

```
The formula
  "not(prop("prop1"))"
is true in states
  {"state2"}
and false in states
  {"state1"}
with winning moves
  state1 -> stay
  state2 -> stay
```

Listing 5: Output of the test case concerning the negation.

```
model
state1 prop1,
state2,
state3 prop2 prop1;
end model
and(prop1, prop2)
```

Listing 6: Small input for the test case concerning the conjunction.

```
The formula
  "and(prop("prop1"),prop("prop2"))"
is true in states
  {"state3"}
and false in states
  {"state1","state2"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
```

Listing 7: Output of the test case concerning the conjunction.

```
model
state1 prop1,
state2,
state3 prop2 prop1;
end model
or(prop1, prop2)
```

Listing 8: Small input for the test case concerning the disjunction.

```
The formula
  "or(prop("prop1"),prop("prop2"))"
is true in states
  {"state1","state3"}
and false in states
  {"state2"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
```

Listing 9: Output of the test case concerning the disjunction.

```
model
state1 prop1,
state2 prop1 prop2;
g1:
  state1 -> state1 state2;
  state2 -> state2, state2 state1;
end func
end model
strat(g1, prop2)
```

Listing 10: Small input for the test case concerning the atomic game.

```
The formula
  "strat(atomic("g1"),prop("prop2"))"
is true in states
  {"state2"}
and false in states
  {"state1"}
with winning moves
  state1 -> stay
  state2 -> {state2}
```

Listing 11: Output of the test case concerning the atomic game.

```
model
state1 prop1,
state2 prop2;
g1:
  state1 -> state1 state2, Empty;
  state2 -> state1, state2 state1;
end func
end model
strat(dual(g1), prop2)
```

Listing 12: Small input for the test case concerning the dual game.

```
The formula
  "strat(dual(atomic("g1")),prop("prop2"))"
is true in states
  {}
and false in states
  {"state1","state2"}
with winning moves
  state1 -> {}
  state2 -> {state1}
```

Listing 13: Output of the test case concerning the dual game.

```
model
state1 prop1,
state2 prop2,
state3 prop1 prop3;
g1:
  state1 -> state2 state3;
  state2 -> state2 state3;
end func
g2:
  state3 -> state2, state1;
  state1 -> state2 state3;
end func
end model
strat(ang_choice(g1, g2), prop1)
```

Listing 14: Small input for the test case concerning the angelic choice.

```
The formula
  "strat(angChoice(atomic("g1"),atomic("g2")),prop("prop1"))"
is true in states
  {"state3"}
and false in states
  {"state1","state2"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> {state1}
```

Listing 15: Output of the test case concerning the angelic choice.

```
model
state1 prop1,
state2 prop1 prop3,
state3 prop2;
g1:
  state2 -> state2 state3;
  state1 -> state2, state3;
  state3 -> state1 state2;
end func
g2:
  state1 -> state2 state3;
  state3 -> state2, state1;
end func
end model
strat(dem_choice(g1, g2), prop1)
```

Listing 16: Small input for the test case concerning the demonic choice.

```
The formula
  "strat(demChoice(atomic("g1"),atomic("g2")),prop("prop1"))"
is true in states
  {"state3"}
and false in states
  {"state1","state2"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
```

Listing 17: Output of the test case concerning the demonic choice.

```
model
state1 prop1,
state2 prop1 prop2,
state3;
g1:
  state1 -> state2 state3;
  state2 -> state1 state2;
end func
g2:
  state2 -> state1;
  state1 -> state1;
end func
end model
strat(seq(g1, g2), prop1)
```

Listing 18: Small input for the test case concerning the composition.

```
The formula
  "strat(seq(atomic("g1"),atomic("g2")),prop("prop1"))"
is true in states
  {"state2"}
and false in states
  {"state1","state3"}
with winning moves
  state1 -> stay
  state2 -> {state1, state2}
  state3 -> stay
```

Listing 19: Output of the test case concerning the composition.

```
model
state0 prop0,
state1 prop1,
state2 prop2,
state3 prop3,
state4 prop4,
state5 prop5,
state6 prop6,
state7 prop7,
state8 prop8,
state9 prop9;
game0:
  state0 -> state1;
  state1 -> state2;
  state2 -> state3;
  state3 -> state4;
  state4 -> state5;
  state5 -> state6;
  state6 -> state7;
  state8 -> state9;
  state9 -> state0;
end func
end model
strat(ang_iter(game0), prop1)
```

Listing 20: Small input for the test case concerning the angelic iteration.

```
The formula
  "strat(angIter(atomic("game0")),prop("prop1"))"
is true in states
  {"state1","state8","state9","state0"}
and false in states
  {"state2","state3","state4","state5","state6","state7"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
  state4 -> stay
  state5 -> stay
  state6 -> stay
  state7 -> stay
  state8 -> {state9}
  state9 -> {state0}
  state0 -> {state1}
```

Listing 21: Output of the test case concerning the angelic iteration.

```
model
state0 prop0,
state1 prop0,
state2 prop0,
state3 prop0,
state4 prop0,
state5 prop0,
state6 ,
state7 prop0,
state8 prop0,
state9 prop0;
game0:
  state0 -> state1;
  state1 -> state2;
  state2 -> state3;
  state3 -> state4;
  state4 -> state5;
  state5 -> state0;
  state6 -> state7;
  state7 -> state8;
  state8 -> state9;
  state9 -> state6;
end func
end model
strat(dem_iter(game0), prop0)
```

Listing 22: Small input for the test case concerning the demonic iteration.

```
The formula
  "strat(demIter(atomic("game0")),prop("prop0"))"
is true in states
  {"state1","state2","state3","state4","state5","state0"}
and false in states
  {"state6","state7","state8","state9"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
  state4 -> stay
  state5 -> stay
  state6 -> stay
  state7 -> stay
  state8 -> stay
  state9 -> stay
  state0 -> stay
```

Listing 23: Output of the test case concerning the demonic iteration.

```
model
state1 prop1 prop2,
state2 prop2,
state3;
end model
strat(ang_test(prop1), prop2)
```

Listing 24: Small input for the test case concerning the angelic test.

```
The formula
  "strat(angTest(prop("prop1")),prop("prop2"))"
is true in states
  {"state1"}
and false in states
  {"state2","state3"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
```

Listing 25: Output of the test case concerning the angelic test.

```
model
state1 prop3,
state2 prop2,
state3 prop1 prop2;
end model
strat(dem_test(prop2), prop1)
```

Listing 26: Small input for the test case concerning the demonic test.

```
The formula
  "strat(demTest(prop("prop2")),prop("prop1"))"
is true in states
  {"state2","state3"}
and false in states
  {"state1"}
with winning moves
  state1 -> stay
  state2 -> stay
  state3 -> stay
```

Listing 27: Output of the test case concerning the demonic test.

B.2 Large Input Example

```
model
s2 p2,
s19,
s28,
s20,
s17 p1 p4 p3 p2,
s10,
s23,
s0 p1 p3 p2,
s3 p3 p4 p2,
s5 p4 p3 p2 p1,
s13 p4 p1 p2 p3,
s12 p2 p3,
s16 p1 p2 p3 p4,
s7 p1 p4 p3 p2,
s11,
s24 p3,
s25 p4 p1,
s1,
s8 p1,
s4 p4 p2 p1,
s9 p1 p2,
s27 p4 p1 p3 p2,
s18 p3 p2,
s26 p4,
s21 p3,
s22 p1 p2 p3 p4,
s15 p3,
s14 p2,
s6 p2 p1,
s29 p3 p4 p1;
g0:
  s28 -> s29,s18;
  s19 -> s10,s19,s18;
  s1 -> s26,s18 s25;
  s13 -> s21 s23 s7,Empty;
  s15 -> s21 s5;
  s21 -> s16,s28,s2;
  s3 -> s5,s10;
  s11 -> s10;
  s10 -> s1 s17;
  s5 -> s16 s2 s4;
  s24 -> Empty;
end func
g2:
  s22 -> s17 s8;
end func
```

```
g1:
  s5 -> s6,s25;
  s21 -> s11 s17 s20 s23 s5 s9,s6;
  s15 -> s19 s6;
  s6 -> s0;
  s11 -> s23 s24 s3 s6,s2;
  s0 -> s8,s16,s28 s7,s11;
end func
g3:
  s20 -> s26;
  s6 -> Empty;
  s16 -> s17;
  s14 -> s12;
  s13 -> s5,s22;
  s23 -> Empty;
  s4 -> s14,s17;
  s8 -> s1 s2;
  s12 -> s15;
  s15 -> s22,s24;
  s11 -> s24;
  s29 -> s8;
  s17 -> s11,s23;
  s28 -> s7;
  s1 -> Empty;
  s5 -> s11 s7;
  s22 -> s2 s3;
  s27 -> s22;
end func
end model
not(or(or(not(not(strat(dem_choice(dem_iter(ang_test(strat(ang_iter(seq(dual(
    ↪ dem_test(strat(dem_choice(ang_choice(g1,g3),ang_iter(g1)),p2))),
    ↪ ang_choice(dem_choice(dual(g1),g2),g0))),not(not(strat(dem_iter(dual(seq
    ↪ (g3,g2))),strat(ang_choice(seq(ang_iter(seq(dual(g3),dem_choice(g3,g2)))
    ↪ ,dem_iter(g1)),dual(dem_iter(g1))),p2))))))),dual(dem_choice(dem_iter(
    ↪ ang_choice(dem_test(strat(g0,p1)),dual(g1))),ang_iter(dual(dem_test(not(
    ↪ p1))))))),and(strat(ang_test(and(not(and(strat(ang_test(strat(g1,p2)),p4
    ↪ ),strat(g0,p2))),strat(dem_choice(g3,dual(ang_iter(dem_choice(dual(g0),
    ↪ dual(dem_choice(g2,g0)))))),p1))),strat(ang_iter(dual(ang_test(strat(
    ↪ ang_iter(dual(g0)),p1)))),p3)),not(not(or(or(strat(ang_choice(ang_iter(
    ↪ dem_test(not(p2))),g2),p3),strat(dual(ang_iter(dem_choice(dem_test(strat
    ↪ (ang_iter(ang_test(not(p2))),p3)),ang_choice(dual(g3),seq(dual(g1),
    ↪ ang_choice(ang_choice(ang_test(not(p4)),dual(g1)),dem_test(not(p3))))))
    ↪ ),strat(ang_iter(g0),p4))),strat(g3,p1)))))))),or(and(strat(ang_choice(
    ↪ dual(g2),dual(ang_iter(g0))),p3),and(strat(ang_iter(dual(g2)),p4),strat(
    ↪ dem_iter(g1),p2))),not(strat(dem_choice(dual(dual(dual(ang_iter(g2)))),
    ↪ g3),p2)))),and(strat(dem_choice(g1,g3),p4),and(strat(dem_iter(dual(g2)),
    ↪ p3),strat(dem_iter(ang_test(strat(g2,p3))),p4)))))
```

Listing 28: Example of a moderately large input file with $|\mathcal{M}| = 100, |S| = 30, |\varphi| = 200$, and $\text{ad}(\varphi) = 1$.

C Time Complexity Verification

C.1 Tables of Execution Time Measurements

Run	Execution time (ms)	Minimum execution time (ms)
1	1441	1441
2	1201	1201
3	1135	1135
4	1131	1131
5	1134	1131
6	1143	1131
7	1126	1126
8	1133	1126
9	1134	1126
10	1153	1126
11	1125	1125
12	1132	1125
13	1138	1125
14	1175	1125
15	1251	1125
16	1588	1125
17	1574	1125
18	1243	1125
19	1241	1125
20	1583	1125
21	1127	1125
22	1134	1125
23	1191	1125
24	1135	1125
25	1140	1125
26	1122	1122
27	1132	1122
28	1181	1122
29	1120	1120
30	1118	1118

Table 10: Model checker execution times and minimum times against the number of runs for identical input ($|\mathcal{M}| = 500, |S| = 100, |\varphi| = 100, \mathrm{ad}(\varphi) = 2$).

$\|S\|$	**Execution time (ms)**
1000	208
5000	1153
10000	2403
15000	3718
20000	4577
25000	5839
30000	6811
35000	8514
40000	10985
45000	10989
50000	12609
55000	14704
60000	14646
65000	17412
70000	17772
75000	18811
80000	20654
85000	21295
90000	23748
95000	24426
100000	28474
105000	31868
110000	32701
115000	36195
120000	37057
125000	38031
130000	39531
135000	42660
140000	43157
145000	46277
150000	48583

Table 11: Model checker execution times against varying state counts for experiment 1.

$\|\varphi\|$	**Execution time (ms)**
534	206
573	214
1109	407
1184	455
1670	572
1816	647
1841	597
1894	610
2083	1163
3120	1219
4712	1709
6294	2058
6584	2196
6852	2309
7459	4221
9897	3799
10371	3551
10708	5568
11098	4455
13253	4300
14049	4647
27226	9944
28144	10345
34554	12294
39832	15578
41440	16280
50069	18773
57871	21723
62465	22779
65458	23448
79052	30261
125274	51555
168370	68028
308792	130739

Table 12: Model checker execution times against varying formula sizes for experiment 2.

ad(φ), **angelic end**	**Execution time (ms)**
6	13
7	15
8	17
9	24
10	30
11	55
12	69
13	161
14	223
15	618
16	878
17	2562
18	3685
19	11281
20	16197
21	52834

Table 13: Model checker execution times against varying formula alternation depths, where the innermost fixpoint operator is an angelic iteration for experiment 3.

ad(φ), **demonic end**	**Execution time (ms)**
6	12
7	14
8	15
9	16
10	18
11	20
12	23
13	25
14	33
15	39
16	61
17	81
18	149
19	208
20	414
21	577
22	1475
23	2131
24	4091
25	6197
26	13454
27	20030
28	44158
29	64806

Table 14: Model checker execution times against varying formula alternation depths, where the innermost fixpoint operator is a demonic iteration for experiment 3.

$\|S\|$	**Execution time (ms)**	$\|S\|$	**Execution time (ms)**	$\|S\|$	**Execution time (ms)**
5	96	170	3977	335	7346
10	205	175	3359	340	6534
15	267	180	1956	345	4003
20	399	185	3700	350	3803
25	423	190	2664	355	4886
30	526	195	4367	360	5008
35	772	200	2892	365	5828
40	770	205	4970	370	4383
45	645	210	4729	375	7714
50	1287	215	3581	380	8582
55	692	220	4351	385	7338
60	1126	225	5235	390	6930
65	1557	230	2602	395	6249
70	1794	235	3579	400	4739
75	1430	240	3346	405	2814
80	1139	245	3573	410	4691
85	1748	250	5269	415	6865
90	1597	255	4866	420	7182
95	1905	260	5078	425	6823
100	1802	265	5048	430	6630
105	2492	270	5378	435	7987
110	1469	275	6867	440	4967
115	2079	280	4284	445	11777
120	2285	285	6649	450	5320
125	1525	290	4585	455	10164
130	2289	295	6723	460	7275
135	2897	300	5749	465	7286
140	2724	305	4382	470	6668
145	1809	310	5396	475	8123
150	3211	315	481	480	3323
155	4456	320	5305	485	9479
160	3382	325	4123	490	7173
165	3504	330	6602	495	4924

Table 15: Model checker execution times against varying state counts for experiment 4.

$\|\mathcal{M}\|$	**Execution time (ms)**	$\|\mathcal{M}\|$	**Execution time (ms)**
5004	931	105091	2183
10008	1003	110095	2809
15012	1028	115099	2451
20018	1300	120088	2431
25021	1400	125083	3248
30030	1312	130106	2963
35027	1476	135130	2711
40027	1361	140114	3038
45043	1323	145137	2914
50047	1487	150118	2938
55031	1384	155142	2461
60048	1961	160125	2638
65055	1433	165133	2851
70058	1506	170139	2492
75055	1850	175156	3758
80049	2417	180138	2978
85055	2240	185173	3400
90075	2655	190166	3280
95091	2101	195141	5110
100071	1731	200146	2766

Table 16: Model checker execution times against varying model sizes for experiment 5.

$\|\varphi\|$	**Execution time (ms)**	$\|\varphi\|$	**Execution time (ms)**
100	497	2600	8868
200	718	2700	9069
300	1153	2800	8464
400	1318	2900	11356
500	1514	3000	11310
600	2143	3100	10673
700	2434	3200	11036
800	2882	3300	10579
900	3333	3400	8926
1000	3546	3500	13777
1100	4150	3600	12485
1200	4712	3700	11849
1300	5103	3800	13030
1400	5239	3900	14209
1500	5078	4000	13014
1600	5828	4100	15266
1700	5767	4200	13657
1800	5420	4300	13937
1900	6204	4400	13947
2000	6713	4500	14954
2100	6171	4600	16719
2200	7977	4700	15056
2300	7420	4800	15530
2400	8624	4900	17374
2500	9346	5000	18787

Table 17: Model checker execution times against varying formula sizes for experiment 6.

$\mathrm{ad}(\varphi)$	**Execution time (ms)**
0	511
1	556
2	646
3	708
4	749
5	786
6	856
7	858
8	756
9	1953
10	1052
11	996
12	1584
13	66386
14	4096
15	41660
16	25325
17	130429

Table 18: Model checker execution times against varying alternation depths with differing innermost fixpoint operators for experiment 7.

ad(φ)	**Execution time (ms)**
0	346
1	551
2	583
3	694
4	589
5	645
6	721
7	751
8	832
9	813
10	922
11	968
12	960
13	833
14	927
15	5005
16	1277
17	15872
18	19706
19	121680
20	14918

Table 19: Model checker execution times against varying alternation depths with an angelic iteration as the innermost fixpoint operator for experiment 7.

ad(φ)	**Execution time (ms)**
0	348
1	526
2	578
3	694
4	711
5	661
6	780
7	781
8	843
9	903
10	1280
11	1295
12	902
13	3900
14	1219
15	1181
16	1164
17	20821
18	25135
19	10138

Table 20: Model checker execution times against varying alternation depths with a demonic iteration as the innermost fixpoint operator for experiment 7.

D Source Code (per module)

D.1 ModelChecker

```
module ModelChecker

import AbstractSyntax;
import AST;
import Consistency;
import DNNF;
import EvalGame;
import IO;
import List;
import Map;
import Set;
import String;
import Util;
import util::Benchmark;
import util::FileSystem;
import util::ShellExec;

// All the parity game solvers that are currently supported by the model checking tool. Up
    ↪ to date with pgsolver as of June 7, 2022.
private list[str] LOCAL_SOLVERS = ["modelchecker", "stratimprloc2", "stratimprlocal"];
private list[str] GLOBAL_SOLVERS = ["bigstep", "dominiondec", "external_solver", "
    ↪ external_solver_univ", "fpiter", "genetic", "guessstrategy",
    "modelchecker", "optstratimprov", "policyiter", "prioprom", "priopromdel", "
    ↪ priopromdeluniv", "priopromplus", "priopromplusuniv",
    "priopromrec", "priopromrecuniv", "priopromuniv", "recursive", "satsolve", "smallprog",
    ↪ "stratimprdisc", "stratimprloc2",
    "stratimprlocal", "stratimprove", "stratimprsat", "succinctsmallprog", "viasat"];

private str DEFAULT_LOCAL_SOLVER = LOCAL_SOLVERS[0];
private str DEFAULT_GLOBAL_SOLVER = GLOBAL_SOLVERS[0];
private loc PG_INPUT_LOCATION = |cwd:///|;
private str PG_INPUT_FILENAME = "pgInput.txt";
public str DEFAULT_CHECKER_INPUT = "default_input.txt";

// Return whether a should appear before b in an ordered list of EvalGamePosData elements
private bool comparePosData(EvalGamePosData a, EvalGamePosData b) = a.id < b.id;

// Sort a set of EvalGamePosData by ID, return a sorted list
private list[EvalGamePosData] sortPosData(set[EvalGamePosData] posData) = sort(posData,
    ↪ comparePosData);

// Return a map from the ids to the positions at which Eloise wins based on the input of the
    ↪ checker and the solver and the output of the solver
private map[int, Position] winningPositionsEloise(AInput checkerInput, set[str] allStates,
    ↪ str pgOutputLineEloise, map[Position, EvalGamePosData] inputPositions) {
  list[Position] initPositions = [pos(state, checkerInput.formula) | state <- allStates];
  str eloisePositions = pgOutputLineEloise[3..size(pgOutputLineEloise)-1];
  list[str] eloisePosStrList = split(", ", eloisePositions);
  try
    toInt(eloisePosStrList[0]);
  catch:
    eloisePosStrList = [];
  set[int] eloisePosSet = {toInt(posStr) | posStr <- eloisePosStrList};
  return (id: p | p <- initPositions, meta(id, _, _, _) <- [inputPositions[p]], id in
    ↪ eloisePosSet);
}

// Return whether p contains a state (not a neighbourhood)
```

```
51 bool hasState(Position p) {
52   switch(p) {
53     case pos(str state, _): return true;
54   }
55   return false;
56 }
57
58 // Return whether p contains a neighbourhood (not a state)
59 bool hasNeighbourhood(Position p) = !hasState(p);
60
61 // Return a map from each state to a description of the winning move/play from that state (a
     ↪ sequence of states and neighbourhoods)
62 map[str, str] getAllWinningMoves(set[str] allStates, AFormula startFormula, map[Position,
     ↪ Position] winningStrategies) {
63   map[str, str] winningMoves = ();
64   for (str state <- allStates) {
65     Position prevPos = pos(state, startFormula);
66     Position nextPos = prevPos;
67     winningMoves[state] = state;
68     while (nextPos in winningStrategies) {
69       prevPos = nextPos;
70       nextPos = winningStrategies[nextPos];
71       if (hasState(prevPos) && hasNeighbourhood(nextPos)) winningMoves[state] += " -\> {<
     ↪ intercalate(", ", nextPos.neighbourhood)>}";
72       if (hasNeighbourhood(prevPos) && hasState(nextPos)) winningMoves[state] += " -\> <
     ↪ nextPos.state>";
73     }
74   }
75   return winningMoves;
76 }
77
78 // Return output of a local model check based on the output of pgsolver
79 private str localOutput(str pgOutput, AInput checkerInput) {
80   list[str] outputLines = split("\n", pgOutput);
81   bool eloiseWins = endsWith(outputLines[7], "0");
82   return "In state \"<checkerInput.state>\", the formula
83          '  \"<checkerInput.formula>\"
84          'is <eloiseWins>.";
85 }
86
87 // Return output of a global model check based on the output of pgsolver
88 private str globalOutput(str pgOutput, AInput checkerInput, map[Position, EvalGamePosData]
     ↪ inputPositions) {
89   list[str] outputLines = split("\n", pgOutput);
90   set[str] allStates = modelStates(checkerInput.model);
91   map[int, Position] eloisePosMap = winningPositionsEloise(checkerInput, allStates,
     ↪ outputLines[8], inputPositions);
92   map[int, Position] allPositionsMap = (inputPositions[p].id: p | p <- inputPositions);
93   list[str] winStratStrings = split(",", outputLines[10][3..size(outputLines[10])-1]) +
     ↪ split(",", outputLines[15][3..size(outputLines[15])-1]);
94   map[int, int] winningStrategiesIds = (toInt(orig): toInt(dest) | string <- winStratStrings
     ↪ , [orig, dest] <- [split("-\>", string)]);
95   map[Position, Position] winningStrategies = (allPositionsMap[id]: allPositionsMap[
     ↪ winningStrategiesIds[id]] | id <- winningStrategiesIds);
96   map[str, str] winningMoves = getAllWinningMoves(allStates, checkerInput.formula,
     ↪ winningStrategies);
97   set[str] trueStates = {state | pos(str state, _) <- range(eloisePosMap)};
98   set[str] falseStates = modelStates(checkerInput.model) - trueStates;
99   return "The formula
100         '  \"<checkerInput.formula>\"
101         'is true in states
102         '  <trueStates>
```

```
103       'and false in states
104       '  <falseStates>
105       'with winning moves<
106       for (str state <- winningMoves) {>
107       '  <winningMoves[state] == state ? "<state> -\> stay" : winningMoves[state]><
108       }>";
109 }
110 
111 // Generate the pgsolver input and write it to the file specified by pgInputLoc
112 // Return whether the model check is local along with the input mapping for pgsolver
113 private tuple[bool, map[Position, EvalGamePosData]] generateSolverInput(AInput checkerInput,
      ↪  loc pgInputLoc) {
114   map[Position, EvalGamePosData] inputPositions = evalGamePositions(checkerInput);
115   str pgInput = solverInput(sortPosData(range(inputPositions)));
116   writeFile(pgInputLoc, pgInput);
117   return <checkerInput.state != "", inputPositions>;
118 }
119 
120 // Return the user chosen solver based on the given arguments
121 // Use the defaults at the top of this file if no solver is given or if the given solver is
      ↪ invalid
122 private str solverToUse(list[str] args, bool isLocalCheck) {
123   str solver = isLocalCheck ? DEFAULT_LOCAL_SOLVER : DEFAULT_GLOBAL_SOLVER;
124   for (idx <- index(args)) {
125     if (args[idx] == "-solver" && size(args) > idx) solver = args[idx + 1];
126   }
127   if ((isLocalCheck && solver notin LOCAL_SOLVERS) || (!isLocalCheck && solver notin
      ↪ GLOBAL_SOLVERS)) {
128     str invalidSolver = solver;
129     solver = isLocalCheck ? DEFAULT_LOCAL_SOLVER : DEFAULT_GLOBAL_SOLVER;
130     println("The parity game solver \"<invalidSolver>\" is not a valid option for <
      ↪ isLocalCheck ? "local" : "global"> model checking.
131         'Using the default solver instead (<solver>).");
132   };
133   return solver;
134 }
135 
136 // Perform a model check on a game model located in the file specified by args[0] relative
      ↪ to the input files folder, or in default_input.txt
137 private str checkModel(list[str] args) {
138   if (anyFalse([exists(|cwd:///src|), exists(|cwd:///input%20files|), exists(|cwd:///
      ↪ pgsolver|)]))
139     return "Could not find files required to run the model checker. Are you working from the
      ↪  root folder (parent of \"input files\")?";
140 
141   loc inputLoc = |cwd:///input%20files| + ((isEmpty(args) || startsWith(args[0], "-")) ?
      ↪ DEFAULT_CHECKER_INPUT : args[0]);
142   if(!exists(inputLoc)) return "The input file could not be found.";
143 
144   // Turn checker input into input for a parity game solver
145   AInput checkerInput = parseInput(inputLoc);
146 
147   checkerInput.formula = dnnf(checkerInput.formula);
148   if (checkerInput == empty() || !isConsistent(checkerInput)) return "";
149   tuple[bool isLocalCheck, map[Position, EvalGamePosData] positions] pgInput =
      ↪ generateSolverInput(checkerInput, PG_INPUT_LOCATION + PG_INPUT_FILENAME);
150 
151   str solver = solverToUse(args, pgInput.isLocalCheck);
152   str pgOutput = exec("pgsolver/bin/pgsolver",
153              args=(pgInput.isLocalCheck ? ["-local", solver, "0"] : ["-global", solver]) + [
      ↪ PG_INPUT_FILENAME],
154              workingDir=resolveLocation(|cwd:///|));
```

```
155   str checkerOutput = pgInput.isLocalCheck ? localOutput(pgOutput, checkerInput):
      ↪ globalOutput(pgOutput, checkerInput, pgInput.positions);
156   return checkerOutput;
157 }
158
159 public void main(list[str] args) = println(checkModel(args));
160
```

D.2 EvalGame

```
1  module EvalGame
2
3  import AbstractSyntax;
4  import Boolean;
5  import List;
6  import Set;
7  import Util;
8
9  // An evaluation game position as defined in https://link.springer.com/chapter
       ↪ /10.1007/978-3-319-73579-5_8
10 data Position = pos(str state, AFormula formula) | pos(list[str state] neighbourhood,
       ↪ AFormula formula);
11
12 // Metadata about a position that is required for the parity game solver input
13 data EvalGamePosData = meta(int id, int prio, bool owner, list[int] successors);
14
15 // Generate the input for a parity game solver based on position metadata
16 public str solverInput(list[EvalGamePosData] metaData) {
17   list[str] lines = ["<meta.id> <meta.prio> <toInt(meta.owner)><succ>;"
18     | meta <- metaData, succ <- [isEmpty(meta.successors) ? "" : " <intercalate(",", meta.
       ↪ successors)>"]];
19   return "parity <size(metaData) - 1>;\n" + intercalate("\n", lines);
20 }
21
22 // Return a mapping from the evaluation game position to unique integer identifiers
23 public map[Position, EvalGamePosData] evalGamePositions(AInput input) {
24   AFormula formula = input.formula;
25   AGameModel model = input.model;
26   set[str] initialStates = isLocalCheck(input) ? {input.state} : modelStates(input.model);
27   map[Position, EvalGamePosData] found = ();
28   int nextId = 0;
29   map[str, set[str]] statePropMap = (state: toSet(props) | stateDef(state, props) <- model.
       ↪ stateDefs);
30   for (str state <- initialStates) {
31     if (pos(state, formula) notin found)
32       <nextId, found> = registerPositions(pos(state, formula), found, nextId, statePropMap,
       ↪ model.nbhfs);
33   }
34   return found;
35 }
36
37 // Return the number of fixpoint operators nested in a formula
38 private int numFixpointOperators(AGame game) {
39   int count = 0;
40   visit (game) {
41     case angIter(_): count += 1;
42     case demIter(_): count += 1;
43   }
44   return count;
45 }
46
```

```
47 // Return the priority of a position in an evaluation game
48 private int parity(pos(_, strat(AGame game, _))) {
49   switch (game) {
50     case angIter(_): return 2*numFixpointOperators(game) + 1;
51     case demIter(_): return 2*numFixpointOperators(game);
52   }
53   return 0;
54 }
55 private int parity(_) = 0;
56 
57 // Return the neighboorhouds of a state based on a neighbourhood function
58 private list[list[str]] neighbourhoodsOf(str state, ANeighbourhoodFunc nbhf) {
59   for (AStateMap stateMap <- nbhf.stateMaps) {
60     if(state == stateMap.state) return [states | nbh(states) <- stateMap.neighbourhoods];
61   }
62   return [];
63 }
64 
65 // Return the neighbourhood function corresponding to the atomic game given
66 // Precondition: nbhfs contains exactly one neighbourhood function corresponding to
     ↪ atomicGame
67 private ANeighbourhoodFunc nbhFuncOf(str atomicGame, list[ANeighbourhoodFunc] nbhfs) {
68   for (ANeighbourhoodFunc nbhf <- nbhfs) {
69     if (atomicGame == nbhf.atomicGame) return nbhf;
70   }
71   return nbhf(atomicGame, []);
72 }
73 
74 // Return whether Abelard owns a position and the neighbouring positions
75 // When it does not matter who owns the position, false is returned for the owner (last 7
     ↪ cases)
76 private tuple[bool, list[Position]] ownerAndNeighbours(Position p, map[str, set[str]]
     ↪ stateDefs, list[ANeighbourhoodFunc] nbhfs) {
77   switch (p) {
78     case pos(str state, prop(name)):
79       if (name in stateDefs[state])
80         return <true, []>;
81       else
82         return <false, []>;
83     case pos(str state, not(prop(name))):
84       if (name in stateDefs[state])
85         return <false, []>;
86       else
87         return <true, []>;
88     case pos(str state, and(f1, f2)): return <true, [pos(state, f1), pos(state, f2)]>;
89     case pos(str state, or(f1, f2)): return <false, [pos(state, f1), pos(state, f2)]>;
90     case pos(str state, strat(atomic(g), f)):
91       return <false, [pos(nbh, strat(atomic(g), f)) | nbh <- neighbourhoodsOf(state,
     ↪ nbhFuncOf(g, nbhfs))]>;
92     case pos(list[str] nbh, strat(atomic(_), f)): return <true, [pos(state, f) | state <-
     ↪ nbh]>;
93     case pos(str state, strat(dual(atomic(g)), f)):
94       return <true, [pos(nbh, strat(dual(atomic(g)), f)) | nbh <- neighbourhoodsOf(state,
     ↪ nbhFuncOf(g, nbhfs))]>;
95     case pos(list[str] nbh, strat(dual(atomic(_)), f)): return <false, [pos(state, f) |
     ↪ state <- nbh]>;
96     case pos(str state, strat(seq(g1, g2), f)): return <false, [pos(state, strat(g1, strat(
     ↪ g2, f)))]>;
97     case pos(str state, strat(angChoice(g1, g2), f)): return <false, [pos(state, or(strat(g1
     ↪ , f), strat(g2, f)))]>;
98     case pos(str state, strat(demChoice(g1, g2), f)): return <false, [pos(state, and(strat(
     ↪ g1, f), strat(g2, f)))]>;
```

```
99      case pos(str state, strat(angIter(g), f)): return <false, [pos(state, or(f, strat(g,
        ↪ strat(angIter(g), f))))]>;
100     case pos(str state, strat(demIter(g), f)): return <false, [pos(state, and(f, strat(g,
        ↪ strat(demIter(g), f))))]>;
101     case pos(str state, strat(angTest(f1), f2)): return <false, [pos(state, and(f1, f2))]>;
102     case pos(str state, strat(demTest(f1), f2)): return <false, [pos(state, or(f1, f2))]>;
103   }
104   return <false, []>;
105 }
106
107 // Register pos and all positions reachable from pos in found with their metadata
108 // Return found, along with the successor of the highest id currently in use in found
109 private tuple[int, map[Position, EvalGamePosData]] registerPositions(Position pos, map[
      ↪ Position, EvalGamePosData] found, int nextId, map[str, set[str]] stateDefs, list[
      ↪ ANeighbourhoodFunc] nbhfs) {
110   <owner, neighbours> = ownerAndNeighbours(pos, stateDefs, nbhfs);
111   EvalGamePosData posData = meta(nextId, parity(pos), owner, []);
112   // Add self-loop to dead-end nodes and set the priority correspondingly to ensure
      ↪ identical results to actual dead-end nodes
113   // This is required, as pgsolver does not support dead-end nodes
114   if(isEmpty(neighbours)) {
115     neighbours = [pos];
116     posData.prio = toInt(!owner);
117   }
118   found[pos] = posData;
119   nextId += 1;
120   for (Position nextPos <- neighbours) {
121     if (nextPos notin found) {
122       found[pos].successors += [nextId];
123       <nextId, found> = registerPositions(nextPos, found, nextId, stateDefs, nbhfs);
124     } else {
125       found[pos].successors += [found[nextPos].id];
126     }
127   }
128   return <nextId, found>;
129 }
130
```

D.3 DNNF

```
1  module DNNF
2
3  import AbstractSyntax;
4
5  // Rewrite rules taken from lemma 2 of
6  // https://link.springer.com/chapter/10.1007/978-3-319-73579-5_8
7  // along with general rewrite rules for obtaining negation normal form in propositional
     ↪ logic
8
9  // dnnf(formula) returns an equivalent formula in Dual Negation Normal Form
10
11 public AFormula dnnf(AFormula orig) {
12   switch (orig) {
13     case not(strat(AGame g, AFormula f)): return strat(dnnf(dual(g)), dnnf(not(f)));
14     case strat(dual(AGame g), AFormula f): return dnnf(not(strat(g, not(f))));
15     case not(strat(AGame g, not(AFormula f))): return strat(dnnf(dual(g)),f);
16     case not(or(AFormula f1, AFormula f2)): return and(dnnf(not(f1)), dnnf(not(f2)));
17     case not(and(AFormula f1, AFormula f2)): return or(dnnf(not(f1)), dnnf(not(f2)));
18     case not(not(AFormula f)): return dnnf(f);
19     case and(AFormula f1, AFormula f2): return and(dnnf(f1), dnnf(f2));
20     case or(AFormula f1, AFormula f2): return or(dnnf(f1), dnnf(f2));
```

```
21      case strat(AGame g, AFormula f): return strat(dnnf(g), dnnf(f));
22      default: return orig;
23    }
24  }
25
26  public AGame dnnf(AGame orig) {
27    switch (orig) {
28      case dual(dual(AGame g)): return dnnf(g);
29      case dual(seq(AGame g1, AGame g2)): return seq(dnnf(dual(g1)), dnnf(dual(g2)));
30      case dual(angChoice(AGame g1, AGame g2)): return demChoice(dnnf(dual(g1)), dnnf(dual(g2)
        ↪ ));
31      case dual(demChoice(AGame g1, AGame g2)): return angChoice(dnnf(dual(g1)), dnnf(dual(g2)
        ↪ ));
32      case dual(angIter(AGame g)): return demIter(dnnf(dual(g)));
33      case dual(demIter(AGame g)): return angIter(dnnf(dual(g)));
34      case dual(angTest(AFormula f)): return demTest(dnnf(not(f)));
35      case dual(demTest(AFormula f)): return angTest(dnnf(not(f)));
36      case angChoice(AGame g1, AGame g2): return angChoice(dnnf(g1), dnnf(g2));
37      case demChoice(AGame g1, AGame g2): return demChoice(dnnf(g1), dnnf(g2));
38      case seq(AGame g1, AGame g2): return seq(dnnf(g1), dnnf(g2));
39      case angIter(AGame g): return angIter(dnnf(g));
40      case demIter(AGame g): return demIter(dnnf(g));
41      case angTest(AFormula f): return angTest(dnnf(f));
42      case demTest(AFormula f): return demTest(dnnf(f));
43      default: return orig;
44    }
45  }
46
```

D.4 Consistency

```
1  module Consistency
2
3  import AbstractSyntax;
4  import AST;
5  import EvalGame;
6  import List;
7  import Set;
8  import Util;
9
10 import IO;
11
12 // Return whether the input only contains references to states that were defined in its
       ↪ state definitions
13 private bool hasUndefinedStates(AInput input) {
14   AGameModel model = input.model;
15   set[str] definedStates = modelStates(model);
16   set[str] mentionedStates = {stateMap.state | nbhf <- model.nbhfs, stateMap <- nbhf.
       ↪ stateMaps}
17       + {state | nbhf <- model.nbhfs, stateMap <- nbhf.stateMaps, nbh <- stateMap.
       ↪ neighbourhoods, state <- nbh.states};
18   if (input.state != "") mentionedStates += input.state;
19   bool consistent = mentionedStates <= definedStates;
20   if (!consistent)
21     println("The given input contains references to states that were not defined in its
       ↪ state definitions. (<intercalate(", ", toList(mentionedStates - definedStates))>)");
22   return !consistent;
23 }
24
25 // Return whether the input formula only contains references to propositions that were
       ↪ defined in the game model
```

```
26 private bool hasUndefinedProps(AInput input) {
27   set[str] definedProps = modelProps(input.model);
28   set[str] mentionedProps = {};
29   visit(input.formula) {
30     case prop(str name): mentionedProps += name;
31   }
32   bool consistent = mentionedProps <= definedProps;
33   if (!consistent)
34       println("The given formula contains references to propositions that were not defined in
    ↪  the game model (<intercalate(", ", toList(mentionedProps - definedProps))>).");
35   return !consistent;
36 }
37
38 // Return whether the input formula only contains atomic games that were defined in the
    ↪ neighbourhood functions of the game model
39 private bool hasUndefinedGames(AInput input) {
40   AGameModel model = input.model;
41   set[str] definedGames = modelGames(model);
42   set[str] mentionedGames = {};
43   visit(input.formula) {
44     case atomic(str name): mentionedGames += name;
45   }
46   bool consistent = mentionedGames <= definedGames;
47   if (!consistent)
48       println("The given formula contains atomic games that were not defined in the
    ↪ neighbourhood functions of the game model (<intercalate(", ", toList(mentionedGames -
    ↪  definedGames))>).");
49   return !consistent;
50 }
51
52 // Return whether there are multiple neighbourhood functions for the same atomic game
53 private bool hasDuplicateGames(list[ANeighbourhoodFunc] nbhfs) {
54   set[str] games = {};
55   for (nbhf(str game, _) <- nbhfs) {
56     if (game in games) {
57       println("The given model contains multiple neighbourhood functions for the same atomic
    ↪  game (<game>).");
58       return true;
59     }
60     games += game;
61   }
62   return false;
63 }
64
65 // Return whether there are multiple mappings for the same state within any neighbourhood
    ↪ function
66 private bool hasDuplicateMappings(list[ANeighbourhoodFunc] nbhfs) {
67   set[str] states = {};
68   for (nbhf(_, list[AStateMap] stateMaps) <- nbhfs) {
69     for (stateMap(str state, _) <- stateMaps) {
70       if (state in states) {
71         println("The given neighbourhood functions contain multiple mappings for the same
    ↪ state (<state>).");
72         return true;
73       }
74       states += state;
75     }
76     states = {};
77   }
78   return false;
79 }
80
```

```
81 // Return whether there are multiple state definitions with the same name
82 private bool hasDuplicateStates(list[AStateDef] stateDefs) {
83   set[str] seenStates = {};
84   for (stateDef(state, _) <- stateDefs) {
85     if (state in seenStates) {
86       println("The given state definitions contain multiple states with the same name (<
   ↪ state>).");
87       return true;
88     }
89     seenStates += state;
90   }
91   return false;
92 }
93
94 // Return whether the input contains any inconsistencies with regards to states, properties
     ↪ or atomic games and print messages correspondingly
95 public bool isConsistent(AInput input) {
96   list[ANeighbourhoodFunc] nbhfs = input.model.nbhfs;
97   if (anyTrue([
98       hasUndefinedStates(input), hasUndefinedProps(input),
99       hasUndefinedGames(input), hasDuplicateGames(nbhfs),
100      hasDuplicateMappings(nbhfs), hasDuplicateStates(input.model.stateDefs)
101      ])) {
102     println("Terminating program");
103     return false;
104   }
105   return true;
106 }
107
```

D.5 AbstractSyntax

```
1 module AbstractSyntax
2
3 // Definition for the abstract syntax of a complete input for the model checker
4 data AInput = input(AGameModel model, AFormula formula, str state)
5       | empty();
6
7 // Definitions for the abstract syntax of a game model
8
9 data AGameModel = model(list[AStateDef] stateDefs , list[ANeighbourhoodFunc] nbhfs);
10 data AStateDef = stateDef(str state, list[str] props);
11
12 data ANeighbourhoodFunc = nbhf(str atomicGame, list[AStateMap] stateMaps);
13 data AStateMap = stateMap(str state, list[ANeighbourhood] neighbourhoods);
14 data ANeighbourhood = nbh(list[str] states);
15
16 // Definitions for the abstract syntax of a formula
17
18 data AFormula
19   = prop(str name)
20   | not(AFormula f)
21   | and(AFormula f1, AFormula f2)
22   | or(AFormula f1, AFormula f2)
23   | strat(AGame g, AFormula f);
24
25 data AGame
26   = atomic(str name)
27   | dual(AGame g)
28   | angChoice(AGame g1, AGame g2)
29   | demChoice(AGame g1, AGame g2)
```

```
30    | seq(AGame g1, AGame g2)
31    | angIter(AGame g)
32    | demIter(AGame g)
33    | angTest(AFormula f)
34    | demTest(AFormula f);
35
```

D.6 ConcreteSyntax

```
1  module ConcreteSyntax
2
3  // Whitespace can be inserted in between any two literals and/or nonterminals in the syntax
       ↪ definitions
4  layout Whitespace = [\t\n\r\ ]*;
5
6  // Regular expression used for all identifiers
7  lexical Identifier  = [a-zA-Z0-9_] !<< [a-zA-Z_][a-zA-Z0-9_]* !>> [a-zA-Z0-9_];
8  lexical State   = Identifier;
9  lexical Proposition = Identifier;
10 lexical AtomicGame  = Identifier;
11
12 // Definition for the concrete syntax of a complete input for the model checker
13 start syntax Input = GameModel Formula State?;
14
15 // Definitions for the concrete syntax of a game model
16
17 start syntax GameModel = "model" StateDefs ";" NeighbourhoodFunc* "end model";
18
19 syntax StateDefs = StateDef RestStateDef*;
20 syntax StateDef = State Proposition*;
21 syntax RestStateDef = "," StateDef;
22
23 syntax NeighbourhoodFunc = AtomicGame ":" StateMap+ "end func";
24 syntax StateMap = State "-\>" Neighbourhoods ";";
25 syntax Neighbourhoods = Neighbourhood RestNeighbourhood*;
26 syntax RestNeighbourhood = "," Neighbourhood;
27 syntax Neighbourhood = State+;
28
29 // Definitions for the concrete syntax of a formula (i.e., the literal input strings)
30
31 start syntax Formula
32   = Proposition
33   | "not(" Formula ")"
34   | "and(" Formula "," Formula ")"
35   | "or(" Formula "," Formula ")"
36   | "strat(" Game "," Formula ")";
37
38 syntax Game
39   = AtomicGame
40   | "dual(" Game ")"
41   | "ang_choice(" Game "," Game ")"
42   | "dem_choice(" Game "," Game ")"
43   | "seq(" Game "," Game ")"
44   | "ang_iter(" Game ")"
45   | "dem_iter(" Game ")"
46   | "ang_test(" Formula ")"
47   | "dem_test(" Formula ")";
48
```

D.7 AST

```
module AST

import AbstractSyntax;
import ConcreteSyntax;
import IO;
import ParseTree;
import Set;

// Concrete definition of an empty neighbourhood. Will be converted to an empty list of
    ↪ states.
private str EMPTY_NBH = "Empty";

// Take the file location of a file containing a complete input and return its corresponding
    ↪ AST
public AInput parseInput(loc l) {
  Tree cst;
  try
    cst = parse(#start[Input], l, allowAmbiguity=true);
  catch ParseError(loc location): {
    loc resolvedLoc = resolveLocation(location);
    println("A parse error occured at <resolvedLoc.uri> from (line, column) (<resolvedLoc.
    ↪ begin.line>, <resolvedLoc.begin.column>) until (<resolvedLoc.end.line>, <resolvedLoc.
    ↪ end.column>). Terminating program.");
    return empty();
  }
  return astInput(cst);
}

// Take a CST of a complete input and return the corresponding AST
private AInput astInput(start[Input] tree) {
  bool ready = false;
  // Get rid of ambiguities at the root before calling tree.top (ambiguities do not have a
    ↪ top)
  do {
    switch (tree) {
      case amb(set[Tree] alternatives):
        tree = getOneFrom(alternatives);
      default:
        ready = true;
    }
  } while (!ready);
  Input input = tree.top;
  return ast(input);
}

// Disambiguate a CST while creating the AST
private value ast(amb(set[Tree] alternatives)) = ast(getOneFrom(alternatives));

// Helpers for turning CST of a complete input into an AST
private AInput ast((Input)`<GameModel m><Formula f><State s>`) = input(ast(m), ast(f), "<s>"
    ↪ );
private AInput ast((Input)`<GameModel m><Formula f>`) = input(ast(m), ast(f), "");

// Helpers for turning CST of a game model into an AST
private AGameModel ast((GameModel)`model<StateDefs defs>;<NeighbourhoodFunc* nbhfs>end model
    ↪ `) = model(ast(defs), [ast(nbhf) | nbhf <- nbhfs]);

private list[AStateDef] ast((StateDefs)`<StateDef def><RestStateDef* rest>`) = [ast(def)] +
    ↪ [ast(other) | other <- rest];
private AStateDef ast((StateDef)`<State s><Proposition* ps>`) = stateDef("<s>", ["<p>" | p
```

```
       ↪ <− ps]);
53 private AStateDef ast((RestStateDef)`,<StateDef def>`) = ast(def);
54
55 private ANeighbourhoodFunc ast((NeighbourhoodFunc)`<AtomicGame g>:<StateMap+ ms>end func`) =
       ↪  nbhf("<g>", [ast(m) | m <− ms]);
56 private AStateMap ast((StateMap)`<State s>−\><Neighbourhood nbh><RestNeighbourhood* rest>;`)
       ↪  = stateMap("<s>", [ast(nbh)] + [ast(other) | other <− rest]);
57 private ANeighbourhood ast((Neighbourhood)`<State+ states>`) {
58   list[str] stateList = ["<s>" | s <− states];
59   if (stateList[0] == EMPTY_NBH) return nbh([]);
60   return nbh(stateList);
61 }
62 private ANeighbourhood ast((RestNeighbourhood)`,<Neighbourhood nbh>`) = ast(nbh);
63
64 // Helpers for turning CST of a formula into an AST
65 private AFormula ast((Formula)`<Identifier p>`) = prop("<p>");
66 private AFormula ast((Formula)`not(<Formula f1>)`) = not(ast(f1));
67 private AFormula ast((Formula)`and(<Formula f1>,<Formula f2>)`) = and(ast(f1), ast(f2));
68 private AFormula ast((Formula)`or(<Formula f1>,<Formula f2>)`) = or(ast(f1), ast(f2));
69 private AFormula ast((Formula)`strat(<Game g1>,<Formula f1>)`) = strat(ast(g1), ast(f1));
70
71 // Helpers for turning CST of a game into an AST
72 private AGame ast((Game)`<Identifier atom>`) = atomic("<atom>");
73 private AGame ast((Game)`dual(<Game g1>)`) = dual(ast(g1));
74 private AGame ast((Game)`ang_choice(<Game g1>,<Game g2>)`) = angChoice(ast(g1), ast(g2));
75 private AGame ast((Game)`dem_choice(<Game g1>,<Game g2>)`) = demChoice(ast(g1), ast(g2));
76 private AGame ast((Game)`seq(<Game g1>,<Game g2>)`) = seq(ast(g1), ast(g2));
77 private AGame ast((Game)`ang_iter(<Game g1>)`) = angIter(ast(g1));
78 private AGame ast((Game)`dem_iter(<Game g1>)`) = demIter(ast(g1));
79 private AGame ast((Game)`ang_test(<Formula f1>)`) = angTest(ast(f1));
80 private AGame ast((Game)`dem_test(<Formula f1>)`) = demTest(ast(f1));module AST
81
82 import AbstractSyntax;
83 import ConcreteSyntax;
84 import IO;
85 import ParseTree;
86 import Set;
87
88 // Concrete definition of an empty neighbourhood. Will be converted to an empty list of
       ↪ states.
89 private str EMPTY_NBH = "Empty";
90
91 // Take the file location of a file containing a complete input and return its corresponding
       ↪  AST
92 public AInput parseInput(loc l) {
93   Tree cst;
94   try
95     cst = parse(#start[Input], l, allowAmbiguity=true);
96   catch ParseError(loc location): {
97     loc resolvedLoc = resolveLocation(location);
98     println("A parse error occured at <resolvedLoc.uri> from (line, column) (<resolvedLoc.
       ↪ begin.line>, <resolvedLoc.begin.column>) until (<resolvedLoc.end.line>, <resolvedLoc.
       ↪ end.column>). Terminating program.");
99     return empty();
100  }
101  return astInput(cst);
102 }
103
104 // Take a CST of a complete input and return the corresponding AST
105 private AInput astInput(start[Input] tree) {
106   bool ready = false;
107   // Get rid of ambiguities at the root before calling tree.top (ambiguities do not have a
```

```
    ↪ top)
108   do {
109     switch (tree) {
110       case amb(set[Tree] alternatives):
111         tree = getOneFrom(alternatives);
112       default:
113         ready = true;
114     }
115   } while (!ready);
116   Input input = tree.top;
117   return ast(input);
118 }
119 
120 // Disambiguate a CST while creating the AST
121 private value ast(amb(set[Tree] alternatives)) = ast(getOneFrom(alternatives));
122 
123 // Helpers for turning CST of a complete input into an AST
124 private AInput ast((Input)`<GameModel m><Formula f><State s>`) = input(ast(m), ast(f), "<s>"
    ↪ );
125 private AInput ast((Input)`<GameModel m><Formula f>`) = input(ast(m), ast(f), "");
126 
127 // Helpers for turning CST of a game model into an AST
128 private AGameModel ast((GameModel)`model<StateDefs defs>;<NeighbourhoodFunc* nbhfs>end model
    ↪ `) = model(ast(defs), [ast(nbhf) | nbhf <- nbhfs]);
129 
130 private list[AStateDef] ast((StateDefs)`<StateDef def><RestStateDef* rest>`) = [ast(def)] +
    ↪ [ast(other) | other <- rest];
131 private AStateDef ast((StateDef)`<State s><Proposition* ps>`) = stateDef("<s>", ["<p>" | p
    ↪ <- ps]);
132 private AStateDef ast((RestStateDef)`,<StateDef def>`) = ast(def);
133 
134 private ANeighbourhoodFunc ast((NeighbourhoodFunc)`<AtomicGame g>:<StateMap+ ms>end func`) =
    ↪  nbhf("<g>", [ast(m) | m <- ms]);
135 private AStateMap ast((StateMap)`<State s>-\><Neighbourhood nbh><RestNeighbourhood* rest>;`)
    ↪  = stateMap("<s>", [ast(nbh)] + [ast(other) | other <- rest]);
136 private ANeighbourhood ast((Neighbourhood)`<State+ states>`) {
137   list[str] stateList = ["<s>" | s <- states];
138   if (stateList[0] == EMPTY_NBH) return nbh([]);
139   return nbh(stateList);
140 }
141 private ANeighbourhood ast((RestNeighbourhood)`,<Neighbourhood nbh>`) = ast(nbh);
142 
143 // Helpers for turning CST of a formula into an AST
144 private AFormula ast((Formula)`<Identifier p>`) = prop("<p>");
145 private AFormula ast((Formula)`not(<Formula f1>)`) = not(ast(f1));
146 private AFormula ast((Formula)`and(<Formula f1>,<Formula f2>)`) = and(ast(f1), ast(f2));
147 private AFormula ast((Formula)`or(<Formula f1>,<Formula f2>)`) = or(ast(f1), ast(f2));
148 private AFormula ast((Formula)`strat(<Game g1>,<Formula f1>)`) = strat(ast(g1), ast(f1));
149 
150 // Helpers for turning CST of a game into an AST
151 private AGame ast((Game)`<Identifier atom>`) = atomic("<atom>");
152 private AGame ast((Game)`dual(<Game g1>)`) = dual(ast(g1));
153 private AGame ast((Game)`ang_choice(<Game g1>,<Game g2>)`) = angChoice(ast(g1), ast(g2));
154 private AGame ast((Game)`dem_choice(<Game g1>,<Game g2>)`) = demChoice(ast(g1), ast(g2));
155 private AGame ast((Game)`seq(<Game g1>,<Game g2>)`) = seq(ast(g1), ast(g2));
156 private AGame ast((Game)`ang_iter(<Game g1>)`) = angIter(ast(g1));
157 private AGame ast((Game)`dem_iter(<Game g1>)`) = demIter(ast(g1));
158 private AGame ast((Game)`ang_test(<Formula f1>)`) = angTest(ast(f1));
159 private AGame ast((Game)`dem_test(<Formula f1>)`) = demTest(ast(f1));
160 
```

D.8 Util

```
module Util

import AbstractSyntax;

// Return whether any booleans in a list are true or false
public bool anyTrue(list[bool] l) = true in l;
public bool anyFalse(list[bool] l) = false in l;

// Return whether the user would like to perform a local model check based on the input
public bool isLocalCheck(AInput input) = input.state != "";

// modelXXX returns the set of XXXs that were defined in the game model
public set[str] modelStates(AGameModel model) = {state | stateDef(state, _) <- model.
    ↪ stateDefs};
public set[str] modelProps(AGameModel model) = {prop | stateDef(_, props) <- model.stateDefs
    ↪ , prop <- props};
public set[str] modelGames(AGameModel model) = {game | nbhf(game, _) <- model.nbhfs};

```

D.9 ComplexityAnalysis

```
module ComplexityAnalysis

import AbstractSyntax;
import AST;
import IO;
import List;
import ModelChecker;
import Util;
import util::Math;
import Set;
import String;

private data Predecessor = angelic() | demonic() | neither();

// https://link.springer.com/article/10.1023/A:1027354826364
// Return the size of a formula (the number of symbols it contains)
private int formulaSize(AFormula formula) {
  int size = 0;
  visit(formula) {
    case str _: ;
    case _: size += 1;
  }
  return size;
}

// https://link.springer.com/article/10.1023/A:1027354826364
// Return the size of a game model (states + neighbourhoods)
private int modelSize(AGameModel model) {
  int numStates = size(modelStates(model));
  try
    return numStates + sum([size(stateList) | nbhf <- model.nbhfs, stateMap <- nbhf.
    ↪ stateMaps, nbh(stateList) <- stateMap.neighbourhoods]);
  catch EmptyList(): return numStates;
}

// https://link.springer.com/article/10.1023/A:1027354826364
// Return a list of alternation depths that are nested in a game, the maximum of which is
    ↪ the alternation depth of the whole game
```

```
37 private list[int] altDepths(Predecessor predecessor, AGame game, list[int] depths) {
38   switch(game) {
39     case atomic(_): return depths;
40     case dual(g): return altDepths(predecessor, g, depths);
41     case angChoice(g1, g2): return altDepths(predecessor, g1, [head(depths)]) + altDepths(
     ↪ predecessor, g2, [head(depths)]) + tail(depths);
42     case demChoice(g1, g2): return altDepths(predecessor, g1, [head(depths)]) + altDepths(
     ↪ predecessor, g2, [head(depths)]) + tail(depths);
43     case seq(g1, g2): return altDepths(predecessor, g1, [head(depths)]) + altDepths(
     ↪ predecessor, g2, [head(depths)]) + tail(depths);
44     case angIter(g): {
45       if (predecessor == angelic()) return altDepths(predecessor, g, depths);
46       return altDepths(angelic(), g, [head(depths) + 1] + tail(depths));
47     }
48     case demIter(g): {
49       if (predecessor == demonic()) return altDepths(predecessor, g, depths);
50       return altDepths(demonic(), g, [head(depths) + 1] + tail(depths));
51     }
52     case angTest(f): return altDepths(neither(), f, [0]) + depths;
53     case demTest(f): return altDepths(neither(), f, [0]) + depths;
54   }
55 }
56
57 // https://link.springer.com/article/10.1023/A:1027354826364
58 // Return a list of alternation depths that are nested in a formula, the maximum of which is
     ↪ the alternation depth of the whole formula
59 private list[int] altDepths(Predecessor predecessor, AFormula formula, list[int] depths) {
60   switch(formula) {
61     case prop(_): return depths;
62     case not(f): return altDepths(predecessor, f, depths);
63     case and(f1, f2): return altDepths(predecessor, f1, [head(depths)]) + altDepths(
     ↪ predecessor, f2, [head(depths)]) + tail(depths);
64     case or(f1, f2): return altDepths(predecessor, f1, [head(depths)]) + altDepths(
     ↪ predecessor, f2, [head(depths)]) + tail(depths);
65     case strat(g, f): return altDepths(predecessor, f, [head(depths)]) + altDepths(
     ↪ predecessor, g, [head(depths)]) + tail(depths);
66   }
67 }
68
69 // Return the alternation depth of a formula
70 private int alternationDepth(AFormula formula) {
71   return max(altDepths(neither(), formula, [0]));
72 }
73
74 // Print some statistics about an input file that are relevant to time complexity analysis
75 public void reportComplexityParams(list[str] args) {
76   if (anyFalse([exists(|cwd:///bin|), exists(|cwd:///src|), exists(|cwd:///input%20files|),
     ↪ exists(|cwd:///pgsolver|)])) {
77     println("Could not find files required to run the model checker. Are you working from
     ↪ the root folder (parent of \"input files\")?");
78     return;
79   }
80   loc inputLoc = |cwd:///input%20files| + ((isEmpty(args) || startsWith(args[0], "-")) ?
     ↪ DEFAULT_CHECKER_INPUT : args[0]);
81   if(!exists(inputLoc)) {
82     println("The input file could not be found.");
83     return;
84   }
85   AInput input = parseInput(inputLoc);
86   println(
87     "Model size:        <modelSize(input.model)>
88     'Formula size:      <formulaSize(input.formula)>
```

```
89      'Alternation depth: <alternationDepth(input.formula)>
90      'Number of states:  <size(modelStates(input.model))>"
91    );
92  }
93
```