Verifying Featured Transition Systems using Variability Parity Games

Sjef van Loo

August 15, 2019

Contents

Part I
Verifying featured transition systems

1 Introduction

Model verification techniques can be used to improve the quality of software. These techniques require the behaviour of the software to be modelled, after which the model can checked to verify that it behaves conforming to some requirement. Different languages are proposed and well studied to express these requirements, examples are LTL, CTL, CTL* and μ-calculus (TODO: cite). Once the behaviour is modelled and the requirement is expressed in some language we can use modal checking techniques to determine if the model satisfies the requirement.

These techniques are well suited to model and verify the behaviour of a single software product. However software systems can be designed to have certain parts enabled or disabled. This gives rise to many software products that all behave very similar but not identical, such a collection is often called a *product family*. The differences between the products in a product family is called the *variability* of the family. A family can be verified by using the above mentioned techniques to verify every single product independently. However this approach does not use the similarities in behaviour of these different products, an approach that would make use of the similarities could potentially be a lot more efficient.

Labelled transition systems (LTSs) are often used to model the behaviour of a system, while it can model behaviour well it can't model variability. Efforts to also model variability include I/O automata, modal transition systems and *featured transition systems* (FTSs) (TODO: cite). Specifically the latter is well suited to model all the different behaviours of the software products as well as the variability of the entire system in a single model.

Efforts have been made to verify requirements for entire FTSs, as well as to be able to reason about features. Notable contributions are fLTL, fCTL and fNuSMV (TODO: cite). However, as far as we know, there is no technique to verify an FTS against a μ-calculus formula. Since the modal μ-calculus is very expressive, it subsumes other temporal logics like LTL, CTL and CTL*, this is desired. In this thesis we will introduce a technique to do this. We first look at LTSs, the modal μ-calculus and FTSs. Next we will look at an existing technique to verify an LTS, namely solving *parity games*, as well as show how this technique can be used to verify an FTS by verifying every software product it describes independently. An extension to this technique is then proposed, namely solving *variability parity games*. We will formally define variability parity games and prove that solving them can be used to verify FTSs.

2 Verifying transition systems

We first look at labelled transition systems (LTSs) and the modal μ-calculus and what it means to verify an LTS. The definitions below are derived from [1].

Definition 2.1. *A labelled transition system (LTS) is a tuple $M = (S, Act, trans, s_0)$, where:*

- *S is a set of states,*
- *Act a set of actions,*
- *$trans \subseteq S \times Act \times S$ is the transition relation with $(s, a, s') \in trans$ denoted by $s \xrightarrow{a} s'$,*
- *$s_0 \in S$ is the initial state.*

Consider the example in figure 1 (directly taken from [2]) of a coffee machine where we have two actions: ins (insert coin) and std (get standard sized coffee).

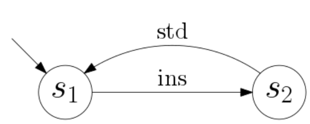


Figure 1: Coffee machine LTS C

Definition 2.2. *A modal μ-calculus formula over the set of actions Act and a set of variables $\mathcal{X}$ is defined by*

$$\varphi = \top \mid \bot \mid X \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X.\varphi \mid \nu X.\varphi$$

with $a \in Act$ and $X \in \mathcal{X}$.

We don't include negations in the language because negations can be pushed inside to the propositions, ie. the $\top$ and $\bot$ elements.

The modal μ-calculus contains boolean constants $\top$ and $\bot$, propositional operators $\vee$ and $\wedge$, modal operators $\langle\rangle$ and $[]$ and fixpoint operators μ and ν. A formula is closed when variables only occur in the scope of a fixpoint operator for that variable.

A modal μ-calculus formula can be interpreted with an LTS, this results in a set of states for which the formula holds.

Definition 2.3. *For LTS $(S, Act, trans, s_0)$ we inductively define the interpretation of a modal μ-calculus formula φ, notation $[\![\varphi]\!]^\eta$, where $\eta : \mathcal{X} \to \mathcal{P}(S)$ is a logical variable valuation, as a set of states where φ is valid, by:*

$$\begin{aligned}
[\![\top]\!]^\eta &= S \\
[\![\bot]\!]^\eta &= \emptyset \\
[\![\varphi_1 \wedge \varphi_2]\!]^\eta &= [\![\varphi_1]\!]^\eta \cap [\![\varphi_2]\!]^\eta \\
[\![\varphi_1 \vee \varphi_2]\!]^\eta &= [\![\varphi_1]\!]^\eta \cup [\![\varphi_2]\!]^\eta \\
[\![\langle a\rangle\varphi]\!]^\eta &= \{s \in S | \exists_{s' \in S} s \xrightarrow{a} s' \wedge s' \in [\![\varphi]\!]^\eta\} \\
[\![[a]\varphi]\!]^\eta &= \{s \in S | \forall_{s' \in S} s \xrightarrow{a} s' \implies s' \in [\![\varphi]\!]^\eta\} \\
[\![\mu X.\varphi]\!]^\eta &= \bigcap_{f \subseteq S} \{f | f = [\![\varphi]\!]^{\eta[X:=f]}\} \\
[\![\nu X.\varphi]\!]^\eta &= \bigcup_{f \subseteq S} \{f | f = [\![\varphi]\!]^{\eta[X:=f]}\} \\
[\![X]\!]^\eta &= \eta(X)
\end{aligned}$$

Given closed formula φ, LTS $M = (S, Act, trans, s_0)$ and $s \in S$ iff $s \in [\![\varphi]\!]^\eta$ for M we say that formula φ holds for M in state s and write $(M, s) \models \varphi$. Iff formula φ holds for M in the initial state we say that formula φ holds for M and write $M \models \varphi$.

Again consider the coffee machine example (figure 1) and formula $\varphi = \nu X.\mu Y.([ins]Y \wedge [std]X)$ (taken from [2]) which states that action *std* must occur infinitely often over all runs. Obviously this holds for the coffee machine, therefore we have $C \models \varphi$.

2.1 Featured transition systems

A *featured transition system* (FTS) extends the LTS definition to express variability. It does so by introducing *features* and *products* into the definition. Features are options that can be enabled or disabled for the system. A product is a feature assignments, ie. a set of features that is enabled for that product. Not all products are valid, some features might be mutually exclusive while others features might always be required. To express the relation between features one can use feature diagrams as explained in [3]. Feature diagrams offer a nice way of expressing which feature assignments are valid, however for simplicity we will represent the collection of valid products simply with a set of feature assignments. Finally FTSs guard every transition with a boolean expression over the set of features.

Definition 2.4. *[3] A featured transition system (FTS) is a tuple $M = (S, Act, trans, s_0, N, P, \gamma)$, where:*

- *$S, Act, trans, s_0$ are defined as in an LTS,*
- *N is a non-empty set of features,*
- *$P \subseteq \mathcal{P}(N)$ is a non-empty set of products, ie. feature assignments, that are valid,*
- *$\gamma : trans \to \mathbb{B}(N)$ is a total function, labelling each transition with a boolean expression over the features. A product $p \in \mathcal{P}(N)$ satisfying the boolean expression of transition t is denoted by $p \models \gamma(t)$. The boolean expression that is satisfied by any feature assignment is denoted by $\top$, ie $p \models \top$ for any p.*

  A transition $s \xrightarrow{a} s'$ and $\gamma(s, a, s') = f$ is denoted by $s \xrightarrow{a|f} s'$.

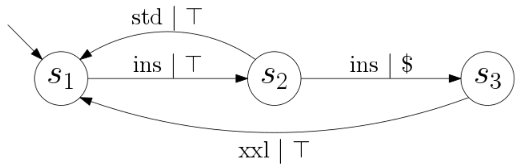


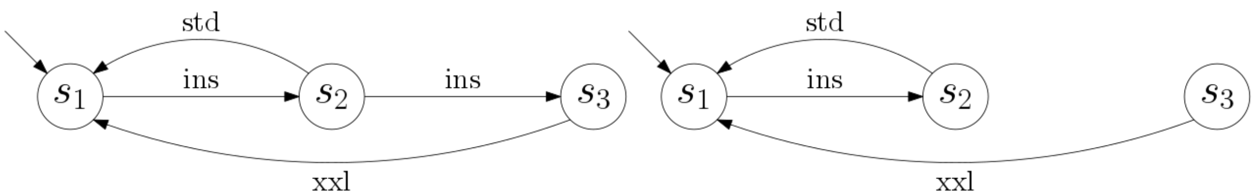
Figure 2: Coffee machine FTS C

Consider the example in figure 2 (directly taken from [2]) which shows an FTS for a coffee machine. For this example we have two features $N = \{\$, €\}$ and two valid products $P = \{\{\$\}, \{€\}\}$.

An FTS expresses the behaviour of multiple products, we can derive the behaviour of a single product by simply removing all the transitions from the FTS for which the product doesn't satisfy the feature expression guarding the transition. We call this a *projection*.

Definition 2.5. *[3] The projection of an FTS $M = (S, Act, trans, s_0, N, P, \gamma)$ to a product $p \in P$, noted $M_{|p}$, is the LTS $M' = (S, Act, trans', s_0)$, where $trans' = \{t \in trans \mid p \models \gamma(t)\}$.*

The coffee machine example can be projected to its two products, which results in the LTSs in figure 3.



(a) C projected to the dollar product: $C_{|\{\$\}}$

(b) C projected to the euro product: $C_{|\{€\}}$

Figure 3: Projections of the coffee machine FTS

2.2 FTS verification question

When verifiying an FTS against a model μ-calculus formula φ, we are trying to answer the question: For which products in the FTS does its projection satisfy φ? Formally, given FTS $M = (S, Act, trans, s_0, N, P, \gamma)$ and modal μ-calculus formula φ we want to find $P_s \subseteq P$ such that:

- for every $p \in P_s$ we have $M_{|p} \models \varphi$ and
- for every $p \in P \backslash P_s$ we have $M_{|p} \not\models \varphi$.

Furthermore a counterexample for every $p \in P \backslash P_s$ is preferred.

3 Verification using parity games

Verifying LTSs against a modal μ-calculus formula can be done by solving a *parity game*. This is done by translating an LTS in combination with a formula to a parity game, the solution of the parity game provides the information needed to conclude if the model satisfies the formula. This relation is depicted in figure 4. This technique is well known and well studied, in this section we will first look at parity games, the translation from LTS and formula to a parity game and finally what we can do with this technique to verify FTS.

3.1 Parity games

Definition 3.1. *[4] A parity game (PG) is a tuple (V, V_0, V_1, E, Ω), where:*

- *$V = V_0 \cup V_1$ and $V_0 \cap V_1 = \emptyset$,*

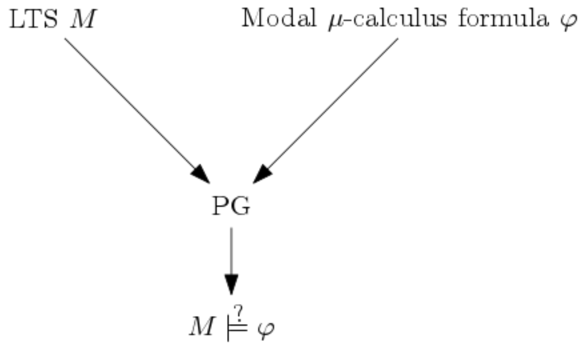


Figure 4: LTS verification using PG

- V_0 *is the set of vertices owned by player* 0*,*
- V_1 *is the set of vertices owned by player* 1*,*
- $E \subseteq V \times V$ *is the edge relation,*
- $\Omega : V \to \mathbb{N}$ *is a priority assignment.*

A parity game is played by players 0 and 1. We write $\alpha \in \{0, 1\}$ to denote an arbitrary player. We write $\overline{\alpha}$ to denote α's opponent, ie. $\overline{0} = 1$ and $\overline{1} = 0$.

A play starts with placing a token on vertex $v \in V$. Player α moves the token if the token is on a vertex owned by α, ie. $v \in V_\alpha$. The token can be moved to $w \in V$, with $(v, w) \in E$. A series of moves results in a sequence of vertices, called a path. For path π we write π_i to denote the i^{th} vertex in path π. A play ends when the token is on vertex $v \in V_\alpha$ and α can't move the token anywhere, in this case player $\overline{\alpha}$ wins the play. If the play results in an infinite path π then we determine the highest priority that occurs infinitely often in this path, formally

$$\max\{p \mid \forall_j \exists_i j < i \wedge p = \Omega(\pi_i)\}$$

If the highest priority is odd then player 1 wins, if it is even player 0 wins.

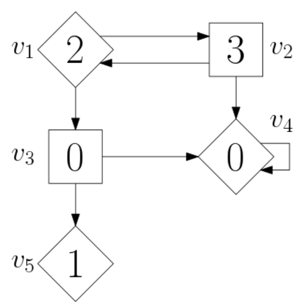


Figure 5: Parity game example

Figure 5 shows an example of a parity game. We usually depict the vertices owned by player 0 by diamonds and vertices owned by player 1 by boxes, the priority is depicted inside the vertices. If the game starts by placing a token on v_1 we can consider the following exemplary paths:

- $\pi = v_1 v_3 v_5$ is won by player 1 since player 0 can't move at v_5.
- $\pi = (v_1 v_2)^\omega$ is won by player 1 since the highest priority occurring infinitely often is 3.
- $\pi = v_1 v_3 (v_4)^\omega$ is won by player 0 since the highest priority occurring infinitely often is 0.

A strategy for player α is a function $\sigma : V^*V_\alpha \to V$ that maps a path ending in a vertex owned by player α to the next vertex. Parity games are positionally determined [4], therefore a strategy $\sigma : V_\alpha \to V$ that maps the current vertex to the next vertex is sufficient.

A strategy σ for player α is winning from vertex v iff any play that results from following σ results in a win for player α. The graph can be divided in two partitions $W_0 \subseteq V$ and $W_1 \subseteq V$, called winning sets. Iff $v \in W_\alpha$ then player α has a winnings strategy from v. Every vertex in the graph is either in W_0 or W_1 [4]. Furthermore finite parity games are decidable [4].

3.2 Creating parity games

A parity game can be created from a combination of an LTS and a modal μ-calculus formula. To do this we introduce some auxiliary definitions regarding the modal μ-calculus.

First we introduce the notion of unfolding, a fixpoint formula $\mu X.\varphi$ can be unfolded resulting in formula φ where every occurrence of X is replaced by $\mu X.\varphi$, denoted by $\varphi[X := \mu X.\varphi]$. A fixpoint formula is equivalent to its unfolding [4], ie. for some LTS $[\![\mu X.\varphi]\!]^\eta = [\![\varphi[X := \mu X.\varphi]]\!]^\eta$. The same holds for the fixpoint operator ν.

Next we define the Fischer-Ladner closure for a closed μ-calculus formula [5, 6]. The Fischer-Ladner closure of φ is the set $FL(\varphi)$ of closed formula's containing at least φ. Furthermore for every formula ψ in $FL(\varphi)$ it holds that for every direct subformula ψ' of ψ there is a formula in $FL(\varphi)$ that is equivalent to ψ'.

Definition 3.2. *The Fischer-Ladner closure of closed μ-calculus formula φ is the smallest set $FL(\varphi)$ satisfying the following constraints:*

- $\varphi \in FL(\varphi)$,
- *if* $\varphi_1 \lor \varphi_2 \in FL(\varphi)$ *then* $\varphi_1, \varphi_2 \in FL(\varphi)$,
- *if* $\varphi_1 \land \varphi_2 \in FL(\varphi)$ *then* $\varphi_1, \varphi_2 \in FL(\varphi)$,
- *if* $\langle a \rangle \varphi' \in FL(\varphi)$ *then* $\varphi' \in FL(\varphi)$,
- *if* $[a]\varphi' \in FL(\varphi)$ *then* $\varphi' \in FL(\varphi)$,
- *if* $\mu X.\varphi' \in FL(\varphi)$ *then* $\varphi'[X := \mu X.\varphi'] \in FL(\varphi)$ *and*
- *if* $\nu X.\varphi' \in FL(\varphi)$ *then* $\varphi'[X := \nu X.\varphi'] \in FL(\varphi)$.

Finally we define alternating depth.

Definition 3.3. *[4] The dependency order on bound variables of φ is the smallest partial order such that $X \leq_\varphi Y$ if X occurs free in $\sigma Y.\psi$. The alternation depth of a μ-variable X in formula φ is the maximal length of a chain $X_1 \leq_\varphi \cdots \leq_\varphi X_n$ where $X = X_1$, variables $X_1, X_3, \ldots$ are μ-variables and variables $X_2, X_4, \ldots$ are ν-variables. The alternation depth of a ν-variable is defined similarly. The alternation depth of formula φ, denoted $adepth(\varphi)$, is the maximum of the alternation depths of the variables bound in φ, or zero if there are no fixpoints.*

Consider the example formula $\varphi = \nu X.\mu Y.([ins]Y \land [std]X)$ which states that for an LTS with $Act = \{ins, std\}$ the action std must occur infinitely often over all runs. Since X occurs free in $\mu Y.([ins]Y \land [std]X)$ we have $adepth(Y) = 1$ and $adepth(X) = 2$. As shown in [4] it holds that formula $\mu X.\psi$ has the same alternation depth as its unfolding $\psi[X := \mu X.\psi]$. Similarly for the greatest fixpoint.

We can now define the transformation from an LTS and a formula to a parity game.

Definition 3.4. *[4] LTS2PG(M, φ) converts LTS $M = (S, Act, trans, s_0)$ and closed formula φ to a PG (V, V_0, V_1, E, Ω).*

A vertex in the parity game is represented by a pair (s, ψ) where $s \in S$ and ψ is a modal μ-calculus formula. We will create a vertex for every state with every formula in the Fischer-Ladner closure of φ. We define the set of vertices:

$$V = S \times FL(\varphi)$$

We create the parity game with the smallest set E such that:

- $V = V_0 \cup V_1$,
- $V_0 \cap V_1 = \emptyset$ *and*
- *for every* $v = (s, \psi) \in V$ *we have:*
  - *If* $\psi = \top$ *then* $v \in V_1$.
  - *If* $\psi = \bot$ *then* $v \in V_0$.
  - *If* $\psi = \psi_1 \vee \psi_2$ *then:*
    $v \in V_0$,
    $(v, (s, \psi_1)) \in E$ *and*
    $(v, (s, \psi_2)) \in E$.
  - *If* $\psi = \psi_1 \wedge \psi_2$ *then:*
    $v \in V_1$,
    $(v, (s, \psi_1)) \in E$ *and*
    $(v, (s, \psi_2)) \in E$.
  - *If* $\psi = \langle a \rangle \psi'$ *then* $v \in V_0$ *and for every* $s \xrightarrow{a} s'$ *we have* $(v, (s', \psi')) \in E$.
  - *If* $\psi = [a]\psi'$ *then* $v \in V_1$ *and for every* $s \xrightarrow{a} s'$ *we have* $(v, (s', \psi')) \in E$.
  - *If* $\psi = \mu X.\psi'$ *then* $(v, (s, \psi'[X := \mu X.\psi'])) \in E$.
  - *If* $\psi = \nu X.\psi'$ *then* $(v, (s, \psi'[X := \nu X.\psi'])) \in E$.

Since the Fischer-Ladner formula's are closed we never get the case $\psi = X$.

Finally we have $\Omega(s, \psi) = \begin{cases} 2\lfloor adepth(X)/2 \rfloor & \text{if } \psi = \nu X.\psi' \\ 2\lfloor adepth(X)/2 \rfloor + 1 & \text{if } \psi = \mu X.\psi' \\ 0 & \text{otherwise} \end{cases}$

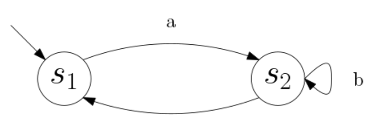


Figure 6: LTS M

Consider LTS M in figure 6 and formula $\varphi = \mu X.([a]X \vee \langle b \rangle \top)$ expressing that on any path reached by a's we can eventually do a b action. We will use this as a working example in the next few sections. The resulting parity game is depicted in figure 7. Solving this parity game results in the following winning sets:

$$\begin{aligned} W_0 = \{&(s_1, \mu X.\phi), \\ &(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top), \\ &(s_1, [a](\mu X.\phi)), \\ &(s_1, \top), \\ &(s_2, \mu X.\phi), \\ &(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top), \\ &(s_2, [a](\mu X.\phi)), \\ &(s_2, \langle b \rangle \top), \\ &(s_2, \top)\} \\ W_1 = \{&(s_1, \langle b \rangle \top)\} \end{aligned}$$

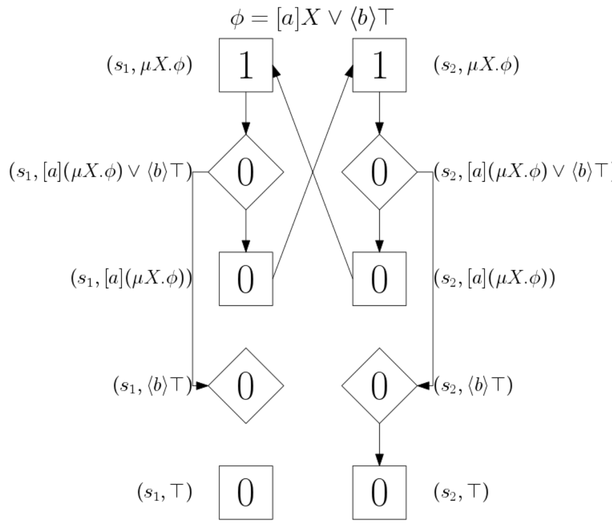


Figure 7: Parity game $LTS2PG(M, \varphi)$

With the strategies σ_0 for player 0 and σ_1 for player 1 being (vertices with one outgoing edge are omitted):

$$\begin{aligned}
\sigma_0 = \{&(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top) \mapsto (s_1, [a](\mu X.\phi)),\\
&(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top) \mapsto (s_2, \langle b \rangle \top)\}\\
\sigma_1 = \{&\}
\end{aligned}$$

State s in LTS M only satisfies φ iff player 0 has a winning strategy from vertex (s, φ). This is formally stated in the following theorem which is proven in [4].

Theorem 3.1. *Given LTS $M = (S, Act, trans, s_0)$, modal μ-calculus formula φ and state $s \in S$ it holds that $(M, s) \models \varphi$ iff $s \in W_0$ for the game $LTS2PG(M, \varphi)$.*

3.3 FTSs and parity games

Using the theory we have seen thus far we can verify FTSs by verifying every projection of the FTS to a valid product. This relation is depicted in the following diagram where Π indicates a projection:

$$\begin{tikzcd}
\text{FTS} \arrow[d, Rightarrow, "\Pi"'] & \\
\text{LTS} \arrow[r, "\varphi"] & \text{PG}
\end{tikzcd}$$

As mentioned before verifying products dependently is potentially more efficient. In the next two sections we define an extension to parity games, namely *variability parity games* (VPGs) which can be used to verify an FTS. We will translate an FTS and a formula into a VPG which solution will provide the information needed to conclude for which products the FTS satisfies the formula.

4 Featured parity games

Before we can define variability parity games we first define *featured parity games* (FPG), featured parity games extend the definition of parity games to capture the variability represented in an FTS. It uses the same concepts as FTSs: features, products and a function that guards edges. In this section we will introduce the definition of FPGs and show that solving them answers the verification questions for FTS: For which products in the FTS does its projection satisfy φ?

First we introduce the definition of an FPG:

Definition 4.1. *A featured parity game (FPG) is a tuple $(V, V_0, V_1, E, \Omega, N, P, \gamma)$, where:*

- $V = V_0 \cup V_1$ *and* $V_0 \cap V_1 = \emptyset$,
- V_0 *is the set of vertices owned by player* 0,
- V_1 *is the set of vertices owned by player* 1,
- $E \subseteq V \times V$ *is the edge relation,*
- $\Omega : V \to \mathbb{N}$ *is a priority assignment,*
- N *is a non-empty set of features,*
- $P \subseteq \mathcal{P}(N)$ *is a non-empty set of products, ie. feature assignments, for which the game can be played,*
- $\gamma : E \to \mathbb{B}(N)$ *is a total function, labelling each edge with a Boolean expression over the features.*

An FPG is played similarly to a PG, however the game is played for a specific product $p \in P$. Player α can only move the token from $v \in V_\alpha$ to $w \in V$ if $(v, w) \in E$ and $p \models \gamma(v, w)$.

A game played for product $p \in P$ results in winnings sets W_0^p and W_1^p, which are defined similar to the W_0 and W_1 winning sets for parity games.

An FPG can simply be projected to a product p by removing the edges that are not satisfied by p.

Definition 4.2. *The projection from FPG $G = (V, V_0, V_1, E, \Omega, N, P, \gamma)$ to a product $p \in P$, noted $G_{|p}$, is the parity game $(V, V_0, V_1, E', \Omega)$ where $E' = \{e \in E \mid p \models \gamma(e)\}$.*

Playing FPG G for a specific product $p \in P$ is the same as playing the PG $G_{|p}$. Any path that is valid in G for p is also valid in $G_{|p}$ and vice versa. Therefore the strategies are also interchangeable, furthermore the winning sets W_α for $G_{|p}$ and W_α^p for G are identical. Since parity games are positionally determined so are FPGs. Similarly, since finite parity games are decidable, so are finite FPGs.

We say that an FPG is solved when the winning sets for every valid product in the FPG are determined.

4.1 Creating featured parity games

An FPG can be created from an FTS in combination with a model μ-calculus formula. We translate an FTS to an FPG by first creating a PG from the transition system as if there were no transition guards, next we apply the same guards to the FPG as are present in the FTS for edges that originate from transitions. The features and valid products in the FPG are identical to those in the FTS.

Definition 4.3. *$FTS2FPG(M, \varphi)$ converts FTS $M = (S, Act, trans, s_0, N, P, \gamma)$ and closed formula φ to FPG $(V, V_0, V_1, E, \Omega, N, P, \gamma')$.*

We have $(V, V_0, V_1, E, \Omega) = LTS2PG((S, Act, trans, s_0), \varphi)$ and

$$\gamma'((s, \psi), (s', \psi')) = \begin{cases} \gamma(s, a, s') & \text{if } \psi = \langle a \rangle \psi' \text{ or } \psi = [a]\psi' \\ \top & \text{otherwise} \end{cases}$$

Consider our working example which we extend to an FTS depicted in figure 8, for this example we have features $N = \{f, g\}$ and products $P = \{\emptyset, \{f\}, \{f, g\}\}$. We can translate this FTS with formula $\varphi = \mu X.([a]X \vee \langle b \rangle \top)$ to an FPG depicted in figure 9. As we can see from the FTS if feature f is enabled and g is disabled then we have an infinite path of a's where b is never enabled, therefore φ doesn't hold for

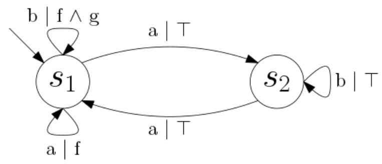


Figure 8: FTS M

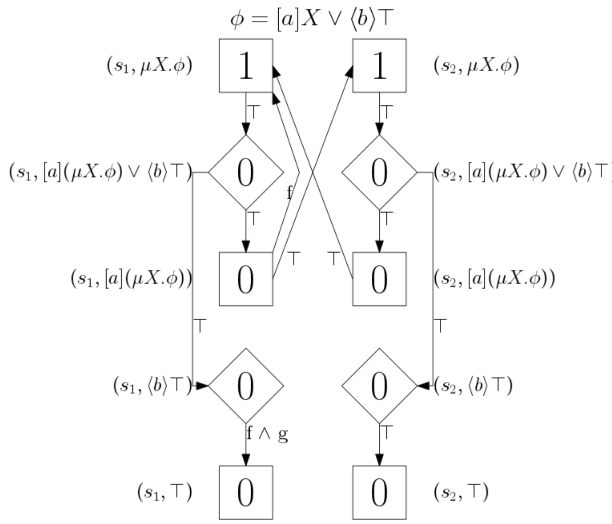


Figure 9: FPG for M and φ

$M_{|\{f\}}$. If g is enabled however we can always do a b so φ holds for $M_{|\{f,g\}}$. As we have seen φ does hold for $M_{|\emptyset}$. For the product $\emptyset$ we have the same winning set as before:

$$
\begin{aligned}
W_0^\emptyset = \{&(s_1, \mu X.\phi),\\
&(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_1, [a](\mu X.\phi)),\\
&(s_1, \top),\\
&(s_2, \mu X.\phi),\\
&(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_2, [a](\mu X.\phi)),\\
&(s_2, \langle b \rangle \top),\\
&(s_2, \top)\}\\
W_1^\emptyset = \{&(s_1, \langle b \rangle \top)\}
\end{aligned}
$$

In the FPG we can see that if f is enabled and g is disabled then player 1 can move the token from $(s_1, [a]X)$ to (s_1, X). This results in player 0 either moving the token to $(s_1, \langle b \rangle \top)$ and losing or an infinite path where

1 occurs infinitely often which is also player 1 wins. For product $\{f\}$ we have winning sets:

$$\begin{aligned}
W_0^{\{f\}} = \{&(s_1, \top),\\
&(s_2, \mu X.\phi),\\
&(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_2, \langle b \rangle \top),\\
&(s_2, \top)\}\\
W_1^{\{f\}} = \{&(s_1, \mu X.\phi),\\
&(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_1, [a](\mu X.\phi)),\\
&(s_1, \langle b \rangle \top),\\
&(s_2, [a](\mu X.\phi))\}
\end{aligned}$$

However if g is also enabled then player 0 wins in $(s_1, \langle b \rangle \top)$, thus giving the following winning sets:

$$\begin{aligned}
W_0^{\{f,g\}} = \{&(s_1, \mu X.\phi),\\
&(s_1, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_1, [a](\mu X.\phi)),\\
&(s_1, \langle b \rangle \top),\\
&(s_1, \top),\\
&(s_2, \mu X.\phi),\\
&(s_2, [a](\mu X.\phi) \vee \langle b \rangle \top),\\
&(s_2, [a](\mu X.\phi)),\\
&(s_2, \langle b \rangle \top),\\
&(s_2, \top)\}\\
W_1^{\{f,g\}} = \{&\}
\end{aligned}$$

In the next section we will show how the winning sets relate to the model verification question.

4.2 FTS verification using FPG

We can create an FPG from an FTS and project it to a product, resulting in a PG, this is shown in the following diagram:

$$\begin{tikzcd}
\text{FTS} \arrow[r, "\varphi"] & \text{FPG} \arrow[d, Rightarrow, "\Pi"] \\
& \text{PG}
\end{tikzcd}$$

Earlier we saw that we could also derive a PG by projecting the FTS to a product and then translation the resulting LTS to a PG, depicted by the following diagram:

$$\begin{tikzcd}
\text{FTS} \arrow[d, Rightarrow, "\Pi"'] & \\
\text{LTS} \arrow[r, "\varphi"] & \text{PG}
\end{tikzcd}$$

We will now show that the resulting parity games are identical.

Theorem 4.1. *Given:*

- *FTS* $M = (S, Act, trans, s_0, N, P, \gamma)$,
- *a closed modal mu-calculus formula* φ,
- *a product* $p \in P$

it holds that the parity games $LTS2PG(M_{|p}, \varphi)$ *and* $FTS2FPG(M, \varphi)_{|p}$ *are identical.*

Proof. Let $G^F = (V^F, V_0^F, V_1^F, E^F, \Omega^F, N, P, \gamma') = FTS2FPG(M, \varphi)$, using definition 4.3, and $G^F_{|p} = (V^F, V_0^F, V_1^F, E^{F'}, \Omega^F)$, using definition 4.2. Furthermore we have $M_{|p} = (S, Act, trans', s_0)$ and we let $G = (V, V_0, V_1, E, \Omega) = LTS2PG(M_{|p}, \varphi)$. We depict the different transition systems and games in the following diagram.

$$\begin{tikzcd}
\text{FTS } M \arrow[rr, "\varphi"] \arrow[d, Rightarrow, "\Pi"'] & & \text{FPG } G^F \arrow[d, Rightarrow, "\Pi"] \\
\text{LTS } M_{|p} \arrow[r, "\varphi"] & \text{PG } G & \text{PG } G^F_{|p}
\end{tikzcd}$$

We will prove that $G = G^F_{|p}$. We first note that game G is created by

$$(V, V_0, V_1, E, \Omega) = LTS2PG((S, Act, trans', s_0), \varphi)$$

and the vertices, edges and priorities of game G^F are created by

$$(V^F, V_0^F, V_1^F, E^F, \Omega^F) = LTS2PG((S, Act, trans, s_0), \varphi)$$

Using the definition of LTS2PG (3.4) we find that the vertices and the priorities only depend on the states in S and the formula φ, since these are identical in the above two statements we immediately get $V = V^F, V_0 = V_0^F, V_1 = V_1^F$ and $\Omega = \Omega^F$. The vertices and priorities don't change when an FTS is projected, therefore $G^F_{|p}$ has the same vertices and priorities as G^F.

Now we are left with showing that $E = E^{F'}$ in order to conclude that that $G = G^F_{|p}$. We will do this by showing $E \subseteq E^{F'}$ and $E \supseteq E^{F'}$.

First let $e \in E$. Note that a vertex in the parity game is represented by a pair of a state and a formula. So we can write $e = ((s, \psi), (s', \psi'))$. To show that $e \in E^{F'}$ we distinguish two cases:

- If $\psi = \langle a \rangle \psi'$ or $\psi = [a]\psi'$ then there exists an $a \in Act$ such that $(s, a, s') \in trans'$. Using the FTS projection definition (2.5) we get $(s, a, s') \in trans$ and $p \models \gamma(s, a, s')$. Using the FTS2FPG definition (4.3) we find that $\gamma'((s, \psi), (s', \psi')) = \gamma(s, a, s')$ and therefore $p \models \gamma'((s, \psi), (s', \psi'))$. Now using the FPG projection definition (4.2) we find $((s, \psi), (s', \psi')) \in E^{F'}$.
- Otherwise the existence of the edge does not depend on the $trans$ parameter and therefore $((s, \psi), (s', \psi')) \in E^{F'}$ if $(s, \psi) \in V^F$, since $V^F = V$ we have $(s, \psi) \in V^F$.

We can conclude that $E \subseteq E^{F'}$, next we will show $E \supseteq E^{F'}$. Let $e = ((s, \psi), (s', \psi')) \in E^{F'}$. We distinguish two cases:

- If $\psi = \langle a \rangle \psi'$ or $\psi = [a]\psi'$ then there exists an $a \in Act$ such that $(s, a, s') \in trans$. Using the FPG projection definition (4.2) we get $p \models \gamma'(s, a, s')$. Using the FTS2FPG definition (4.3) we get $p \models \gamma(s, a, s')$. Using the FTS projection definition (2.5) we get $(s, a, s') \in trans'$ and therefore $((s, \psi), (s', \psi')) \in E$.
- Otherwise the existence of the edge does not depend on the $trans$ parameter and therefore $((s, \psi), (s', \psi')) \in E$ if $(s, \psi) \in V$, since $V^F = V$ we have $(s, \psi) \in V$.

$\square$

Having proven this we can visualize the relation between the different games and transition systems in the following diagram:

$$\begin{tikzcd} \text{FTS} \arrow[r, "\varphi"] \arrow[d, Rightarrow, "\Pi"'] & \text{FPG} \arrow[d, Rightarrow, "\Pi"] \\ \text{LTS} \arrow[r, "\varphi"] & \text{PG} \end{tikzcd}$$

Finally we prove that solving an FTS, ie. finding winning sets for all products, answers the verification question.

Theorem 4.2. *Given:*

- *FTS* $M = (S, Act, trans, s_0, N, P, \gamma)$,
- *closed modal mu-calculus formula* φ,
- *product* $p \in P$ *and*
- *state* $s \in S$

it holds that $(M_{|p}, s) \models \varphi$ *if and only if* $(s, \varphi) \in W_0^p$ *in FTS2FPG*(M, φ).

Proof. The winning set W_α^p is equal to winning set W_α in *FTS2FPG*$(M, \varphi)_{|p}$, for any $\alpha \in \{0, 1\}$, using the FPG definition (4.1). Using theorem 4.1 we find that the game $FTS2FPG(M, \varphi)_{|p}$ is equal to the game $LTS2PG(M_{|p}, \varphi)$, obviously their winning sets are also equal. Using the well studied relation between parity games and LTS verification, stated in theorem 3.1, we know that $(M_{|p}, s) \models \varphi$ if and only if $(s, \varphi) \in W_0$ in game $LTS2PG(M_{|p}, \varphi)$. Winning set W_α^p is equal to W_α, therefore the theorem holds. $\square$

Revisiting our prior example we can see the theorem in action by noting that $M_{|\emptyset} \models \varphi$, $M_{|\{f\}} \not\models \varphi$ and $M_{|\{f,g\}} \models \varphi$. This is reflected by the vertex $(s_1, \mu X.[a]X \vee \langle b \rangle \top)$ being present in $W_0^\emptyset$ and $W_0^{\{f,g\}}$ but not in $W_0^{\{f\}}$.

5 Variability parity games

Next we will introduce *variability parity games* (VPGs). VPGs are very similar to FPGs, however VPGs use configurations instead of features and products to express variability. This gives a syntactically more pleasant representation that is not solely tailored for FTSs. Furthermore in VPGs deadlocks are removed, by doing so VPG plays can only result in infinite paths and no longer in finite paths.

Later we will show the relation between VPGs and FTS verification, which is similar to the relation between FPGs and FTS verification. First we introduce VPGs.

Definition 5.1. *A variability parity game (VPG) is a tuple* $(V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$, *where:*

- $V = V_0 \cup V_1$ *and* $V_0 \cap V_1 = \emptyset$,
- V_0 *is the set of vertices owned by player* 0,
- V_1 *is the set of vertices owned by player* 1,
- $E \subseteq V \times V$ *is the edge relation; we assume that* E *is total, i.e. for all* $v \in V$ *there is some* $w \in V$ *such that* $(v, w) \in E$,
- $\Omega : V \to \mathbb{N}$ *is a priority assignment,*
- $\mathfrak{C}$ *is a non-empty finite set of configurations,*

- $\theta : E \to \mathcal{P}(\mathfrak{C}) \setminus \{0\}$ *is the configuration mapping, satisfying for all* $v \in V$, $\bigcup\{\theta(v,w) \mid (v,w) \in E\} = \mathfrak{C}$.

A VPG is played similarly to a PG, however the game is played for a specific configuration $c \in \mathfrak{C}$. Player α can only move the token from $v \in V_\alpha$ to $w \in V$ if $(v,w) \in E$ and $c \in \theta(v,w)$. Furthermore VPGs don't have deadlocks, therefore every play results in an infinite path.

A game played for configuration $c \in \mathfrak{C}$ results in winning sets W_0^c and W_1^c, which are defined similar to the W_0 and W_1 winning sets for parity games.

Solving a VPG means determining winning sets for every configuration in the VPG.

Definition 5.2. *The projection from VPG* $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ *to a configuration* $c \in \mathfrak{C}$, *noted* $G_{|c}$, *is the parity game* $(V, V_0, V_1, E', \Omega)$ *where* $E' = \{e \in E \mid c \in \theta(e)\}$.

Playing VPG G for a specific configuration $c \in \mathfrak{C}$ is the same as playing the PG $G_{|c}$. Any path that is valid in G for c is also valid in $G_{|c}$ and vice versa. Therefore the strategies are also interchangeable, furthermore the winning sets W_α for $G_{|c}$ and W_α^c for G are identical. Since parity games are positionally determined so are VPGs. Similarly, since finite parity games are decidable, so are finite VPGs.

5.1 Creating variability parity games

We will define a translation from an FPG to a VPG. To do so we use the set of valid products as the set of configurations. Furthermore we make the FPG deadlock free, this is done by creating two losing vertices l_0 and l_1 such that player α loses when the token is in vertex l_α. Any vertex that can't move for a configuration will get an edge that is admissible for that configuration towards one of the losing vertices.

Definition 5.3. *FPG2VPG*(G^F) *converts FPG* $G^F = (V^F, V_0^F, V_1^F, E^F, \Omega^F, N, P, \gamma)$ *to VPG* $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$.

We define $\mathfrak{C} = P$. *We create vertices* l_0 *and* l_1 *and define* $V_0 = V_0^F \cup \{l_0\}$, $V_1 = V_1^F \cup \{l_1\}$ *and* $V = V_0 \cup V_1$.

We construct E *by first making* $E = E^F$ *and adding edges* (l_0, l_0) *and* (l_1, l_1) *to* E. *Simultaneously we construct* θ *by first making* $\theta(e) = \{p \in \mathfrak{C} \mid p \models \gamma(e)\}$ *for every* $e \in E^F$. *Furthermore* $\theta(l_0, l_0) = \theta(l_1, l_1) = \mathfrak{C}$.

Next, for every vertex $v \in V_\alpha$ *with* $\alpha = \{0, 1\}$, *we have* $C = \mathfrak{C} \setminus \bigcup\{\theta(v,w) \mid (v,w) \in E\}$. *If* $C \neq \emptyset$ *then we add* (v, l_α) *to* E *and make* $\theta(v, l_\alpha) = C$. *Finally we have*

$$\Omega(v) = \begin{cases} 1 & \textit{if } v = l_0 \\ 0 & \textit{if } v = l_1 \\ \Omega^F(v) & \textit{otherwise} \end{cases}$$

Again considering our previous working example we can translate the FPG shown in figure 9 to the VPG shown in figure 10. Where c_0 is product $\emptyset$, c_1 is $\{f\}$ and c_2 is $\{f, g\}$.

5.2 FTS verification using VPG

We have shown in theorem 4.2 that we can use an FPG to verify an FTS. Next we will show that a winning set in the FPG M is the subset of the winning set in the VPG *FPG2VPG*(M).

Theorem 5.1. *Given:*

- *FPG* $G^F = (V^F, V_0^F, V_1^F, E^F, \Omega^F, N, P, \gamma)$,
- *product* $p \in P$

we have for winning sets Q_α^p *in* G^F *and* W_α^p *in FPG2VPG*(G^F) *that* $Q_\alpha^p \subseteq W_\alpha^p$ *for any* $\alpha \in \{0, 1\}$.

Proof. Let $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta) =$ *FPG2VPG*(G^F). Consider finite play π that is valid in game G^F for product p. We have for every (π_i, π_{i+1}) in π that $(\pi_i, \pi_{i+1}) \in E^F$ and $p \models \gamma(\pi_i, \pi_{i+1})$. From the *FPG2VPG* definition (5.3) it follows that $(\pi_i, \pi_{i+1}) \in E$ and $p \in \theta(\pi_i, \pi_{i+1})$. So we can conclude that path π is also valid in game G for configuration p. Since the play is finite the winner is determined by the last vertex v in π, player α wins such that $v \in V_{\overline{\alpha}}$. Furthermore we know, because the play is finite, that there exists no $(v,w) \in E^F$ with $p \models \gamma(v,w)$. From this we can conclude that $(v, l_{\overline{\alpha}}) \in E$ and $p \in \theta(v, l_{\overline{\alpha}})$. Vertex $l_{\overline{\alpha}}$ has one

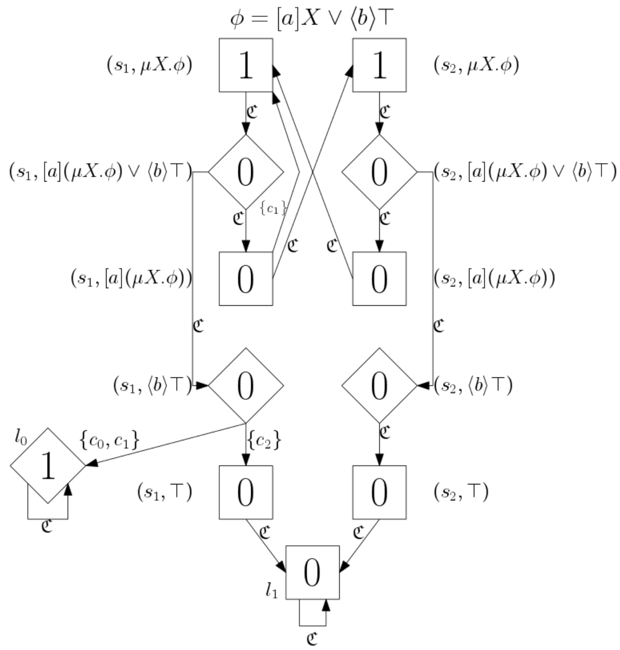


Figure 10: VPG

outgoing edge, namely to itself. So finite play π will in game G^F results in an infinite play $\pi(l_{\overline{\alpha}})^\omega$. Vertex $l_{\overline{\alpha}}$ has a priority with the same parity as player α, so player α wins the infinite play in G for configuration p.

Consider infinite play π that is valid in game G^F for product p. As shown above this play is also valid in game G for configuration p. Since the win conditions of both games are the same the play will result in the same winner.

Consider infinite play π that is valid in game G for configuration p. We distinguish two cases:

- If l_α doesn't occur in π then the path is also valid for game G^F with product p and has the same winner.
- If $\pi = \pi'(l_\alpha)^\omega$ with no occurrence of l_α in π' then the winner is player $\overline{\alpha}$. The path π' is valid for game G^F with product p. Let vertex v be the last vertex of π'. Since $(v, l_\alpha) \in E$ and $p \in \theta(v, l_\alpha)$ we know that there is no $(v, w) \in E^F$ with $p \models \gamma(v, w)$ and that vertex v is owned by player α. So in game G^F player α can't move at vertex v and therefore loses the game (in which case the winner is also $\overline{\alpha}$).

We have shown that every path (finite or infinite) in game G^F with product p can be played in game G with configuration p and that they have the same winner. Furthermore every infinite path in game G with configuration p can be either played as an infinite path or the first part of the path can be played in G^F with product p and they have the same winner. From this we can conclude that the theorem holds. □

We can conclude the diagram depicting the relation between the different games and transition systems:

$$\begin{tikzcd}
\text{FTS} \arrow[r, "\varphi"] \arrow[d, Rightarrow, "\Pi"'] & \text{FPG} \arrow[r] \arrow[d, Rightarrow, "\Pi"] & \text{VPG} \\
\text{LTS} \arrow[r, "\varphi"] & \text{PG} &
\end{tikzcd}$$

Finally we show that solving VPGs, ie. finding the winning sets for all configurations, can be used to verify FTSs.

Theorem 5.2. *Given:*

- *FTS* $M = (S, Act, trans, s_0, N, P, \gamma)$,
- *closed modal mu-calculus formula* φ,
- *product* $p \in P$ *and*
- *state* $s \in S$

it holds that $(M_{|p}, s) \models \varphi$ *if and only if* $(s, \varphi) \in W_0^p$ *in* $\mathit{FPG2VPG}(\mathit{FTS2FPG}(M, \varphi))$.

Proof. Let W_0^p and W_1^p denote the winning sets for game $\mathit{FPG2VPG}(\mathit{FTS2FPG}(M, \varphi))$. And Q_0^p and Q_1^p denote the winning sets for game $\mathit{FTS2FPG}(M, \varphi)$.

Using theorem 4.2 we find that $(M_{|p}, s) \models \varphi$ if and only if $(s, \varphi) \in Q_0^p$. If $(s, \varphi) \in Q_0^p$ then we find by using theorem 5.1 that $(s, \varphi) \in W_0^p$. If $(s, \varphi) \notin Q_0^p$ then $(s, \varphi) \in Q_1^p$ and therefore $(s, \varphi) \in W_1^p$ and $(s, \varphi) \notin W_0^p$. □

Using this theorem we can visualize verification of an FTS in figure 11.

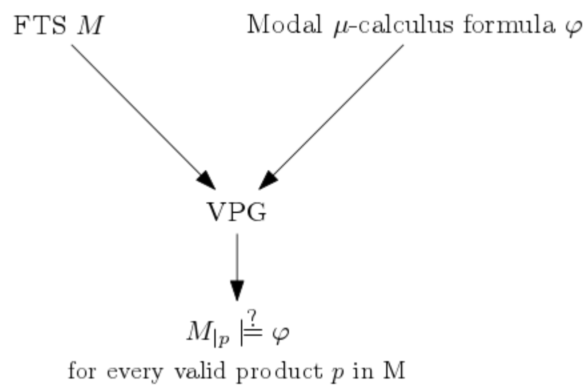


Figure 11: FTS verification using VPG

Part II
Solving variability parity games

6 Introduction

For solving VPGs we distinguish two general approaches, the first approach is to simply project the VPG to the different configurations and solve all the resulting parity games independently. We call this approach *product* based. Alternatively we solve the VPG *family* based where a VPG is solved in its entirety and similarities between the configurations are used to improve performance.

In this next sections we explore family based algorithms, analyse their time complexity and present the results of experiments conducted to test the performance of the different family based algorithms compared to the product based approach. We aim to solve VPGs originating from model verification problems, such VPGs generally have certain properties that a completely random VPG might not have. In general parity games originating from model verification problems have a relatively low number of distinct priorities compared to the number of vertices because new priorities are only introduced when fixed points are nested in the μ-calculus formula. Furthermore the transition guards of featured transition systems are expressed over features. In general these transition guards will be quite simple, specifically excluding or including a small number of features.

7 Preliminary concepts

7.1 Set representation

A set can straightforwardly be represented by a collection containing all the elements that are in the set. We call this an *explicit* representation of a set. We can also represent sets *symbolically* in which case the set of elements is represented by some sort of formula. A typical way to represent a set symbolically is through a boolean formula encoded in a *binary decision diagram* [7, 8]. For example the set $S = \{0, 1, 2, 4, 5, 7\}$ can be expressed by boolean formula:

$$F(x_2, x_1, x_0) = (x_2 \vee \neg x_1 \vee \neg x_0) \wedge (\neg x_2 \vee \neg x_1 \vee x_0)$$

where x_0, x_1 and x_2 are boolean variables. The formula gives the following truth table:

$\mathbf{x_2x_1x_0}$	$\mathbf{F(x_2, x_1, x_0)}$
000	1
001	1
010	1
011	0
100	1
101	1
110	0
111	1

The function F defines set S' in the following way: $S' = \{x_2x_1x_0 \mid F(x_2, x_1, x_0) = 1\}$. As we can see set S' and S represent the same numbers. We can perform set operations on sets represented as boolean functions by performing logical operations on the functions. For example, given boolean formula's f and g representing sets V and W the formula $f \wedge g$ represents set $V \cap W$.

Boolean functions can efficiently be represented in BDDs, for a comprehensive treatment of BDDs we refer to [7, 8]. We will note here that given x boolean variables and two boolean functions encoded as BDDs we can perform binary operations $\vee, \wedge$ on them in $O(2^{2x}) = O(n^2)$ where $n = 2^x$ is the maximum set size that can be represented by x variables [9, 8]. The running time specifically depends on the size of the decision diagrams, in general if the boolean functions are simple then the size of the decision diagram is also small and operations can be performed quickly.

7.1.1 Symbolically representing sets of configurations

For VPGs originating from an FTS the configuration sets are already boolean functions over the features. The formula's guarding the edges in the VPG will generally have relatively simple boolean functions, therefore they are specifically appropriate to represent as BDDs.

A set operation over two explicit sets can be performed in $O(n)$ where n is the maximum size of the sets, this is better than the time complexity of a set operation using BDDs ($O(n^2)$). However if the BDDs are small then the set size can still be large but the set operations can be performed very quickly. This is a trade-off between worst case running time complexity and average actual running time; using a symbolic representation might yield better results if the sets are structured in such a way that the BDDs are small, however its worse case running time complexity will be worse.

7.2 Fixed-point preliminaries

7.2.1 Lattices

The following definition regarding ordering and lattices are taken from [14].

Definition 7.1. *A partial order is a binary relation $x \leq y$ on set S where for all $x, y, z \in S$ we have:*

- *$x \leq x$. (Reflexive)*
- *If $x \leq y$ and $y \leq x$, then $x = y$. (Antisymmetric)*
- *If $x \leq y$ and $y \leq z$, then $x \leq z$. (Transitive)*

Definition 7.2. *A partially ordered set is a set S and a partial order $\leq$ for that set, we denote a partially ordered set by $\langle S, \leq \rangle$.*

Definition 7.3. *Given partially ordered set $\langle P, \leq \rangle$ and subset $X \subseteq P$. An upper bound to X is an element $a \in P$ such that $x \leq a$ for every $x \in X$. A least upper bound to X is an upper bound $a \in P$ such that $a' \leq a$ for every upper bound $a' \in P$ to X.*

The term least upper bound is synonymous with the term supremum.

Definition 7.4. *Given partially ordered set $\langle P, \leq \rangle$ and subset $X \subseteq P$. A lower bound to X is an element $a \in P$ such that $a \leq x$ for every $x \in X$. A greatest lower bound to X is a lower bound $a \in P$ such that $a \leq a'$ for every lower bound $a' \in P$ to X.*

The term greatest lower bound is synonymous with the term infimum.

Definition 7.5. *A lattice is a partially ordered set where any two of its elements have a supremum and an infimum.*

Definition 7.6. *A complete lattice is a partially ordered set in which every subset has a supremum and an infimum.*

Definition 7.7. *A function $f : D \rightarrow D'$ is monotonic, also called order preserving, if for all $x \in D$ and $y \in D$ it holds that if $x \leq y$ then $f(x) \leq f(y)$.*

7.2.2 Fixed-points

Definition 7.8. *Given function $f : D \rightarrow D$ the value $x \in D$ is a fixed point for f if and only if $f(x) = x$.*

Definition 7.9. *Given function $f : D \rightarrow D$ the value $x \in D$ is the least fixed point for f if and only if x is a fixed point for f and every other fixed point for f is greater or equal to x.*

Definition 7.10. *Given function $f : D \rightarrow D$ the value $x \in D$ is the greatest fixed point for f if and only if x is a fixed point for f and every other fixed point for f is less or equal to x.*

The Knaster-Tarski theorem states that least and greatest fixed points exist for some domain and function given that a few conditions hold. The theorem, as written down by Tarski [15], states:

Theorem 7.1 (Knaster-Tarski[15]). *Let*

- $\langle A, \leq \rangle$ *be a complete lattice,*
- f *be an increasing function on* A *to* A*,*
- P *be a the set of all fixpoints of* f*.*

Then the set P *is not empty and the system* $\langle P, \leq \rangle$ *is a complete lattice; in particular we have*

$$\sup P = \sup\{x \mid f(x) \geq x\} \in P$$

and

$$\inf P = \inf\{x \mid f(x) \leq x\} \in P$$

8 Unified parity games

We can create a PG from a VPG by taking all the projections of the VPG, which are PGs, and combining them into one PG by taking the union of them. We call the resulting PG the *unification* of the VPG. A parity game that is the result of a unification is called a *unified PG*, also any total subgame of it will be called a unified PG. A unified PG always has a VPG from which it originated.

Definition 8.1. *Given VPG* $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ *we define the unification of* $\hat{G}$*, denoted as* $\hat{G}_\downarrow$*, as*

$$\hat{G}_\downarrow = \bigcup_{c \in \mathfrak{C}} \hat{G}_{|c}$$

where the union of two PGs is defined as

$$(V, V_0, V_1, E, \Omega) \cup (V', V_0', V_1', E', \Omega') = (V \uplus V', V_0 \uplus V_0', V_1 \uplus V_1', E \uplus E', \Omega \uplus \Omega')$$

We will use the hat decoration $(\hat{G}, \hat{V}, \hat{E}, \hat{\Omega}, \hat{W})$ when referring to a VPG and use no hat decoration when referring to a PG.

Every vertex in game $\hat{G}_\downarrow$ originates from a configuration and an original vertex. Therefore we can consider every vertex in a unification as a pair consisting of a vertex and a configuration, ie. $V = \mathfrak{C} \times \hat{V}$. We can consider edges in a unification similarly, so $E \subseteq (\mathfrak{C} \times \hat{V}) \times (\mathfrak{C} \times \hat{V})$. Note that edges don't cross configurations, ie. for every $((c, \hat{v}), (c', \hat{v}')) \in E$ we have $c = c'$.

If we solve the PG that is the unification of a VPG we have solved the VPG, as shown in the next theorem.

Theorem 8.1. *Given*

- *VPG* $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$*,*
- *some configuration* $c \in \mathfrak{C}$*,*
- *winning sets* $\hat{W}_0^c$ *and* $\hat{W}_1^c$ *for game* $\hat{G}$ *and*
- *winning sets* W_0 *and* W_1 *for game* $\hat{G}_\downarrow$

it holds that

$$(c, \hat{v}) \in W_\alpha \iff \hat{v} \in \hat{W}_\alpha^c, \textit{ for } \alpha \in \{0, 1\}$$

Proof. The bi-implication is equal to the following to implications.

$$(c, \hat{v}) \in W_\alpha \implies \hat{v} \in \hat{W}_\alpha^c, \text{ for } \alpha \in \{0, 1\}$$

and

$$(c, \hat{v}) \notin W_\alpha \implies \hat{v} \notin \hat{W}_\alpha^c, \text{ for } \alpha \in \{0, 1\}$$

Since the winning sets partition the game we have $\hat{v} \notin \hat{W}_\alpha^c \implies \hat{v} \in \hat{W}_{\overline{\alpha}}^c$ (similar for set W). Therefore it is sufficient to prove only the first implication.

Let $(c, \hat{v}) \in W_\alpha$, player α has a strategy to win game $\hat{G}_\downarrow$ from vertex $(c, \hat{v})$. Since $\hat{G}_\downarrow$ is the union of all the projections of $\hat{G}$ we can apply the same strategy to game $\hat{G}_{|c}$ to win vertex $\hat{v}$ as player α. Because we can win $\hat{v}$ in the projection of $\hat{G}$ to c we have $\hat{v} \in \hat{W}_\alpha^c$. $\square$

8.1 Representing unified parity games

Unified PGs have a specific structure because they are the union of PGs that have the same vertices with the same owner and priority. Because they have the same priority we don't actually need to create a new function that is the unification of all the projections, we can simply use the original priority assignment function because the following relation holds:

$$\Omega(c, \hat{v}) = \hat{\Omega}(\hat{v})$$

Similarly we can use the original partition sets $\hat{V}_0$ and $\hat{V}_1$ instead of having the new partition V_0 and V_1 because the following relations holds:

$$(c, \hat{v}) \in V_0 \iff \hat{v} \in \hat{V}_0$$
$$(c, \hat{v}) \in V_1 \iff \hat{v} \in \hat{V}_1$$

So instead of considering unified PG (V, V_0, V_1, E, Ω) we will consider $(V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$.

Next we consider how we represent vertices and edges in a unified PG. A set $X \subseteq (\mathfrak{C} \times \hat{V})$ can be represented as a complete function $f : \hat{V} \to 2^{\mathfrak{C}}$. The set X and function f are equivalent, denoted by the operator $=_\lambda$, iff the following relation holds:

$$(c, \hat{v}) \in X \iff c \in f(\hat{v})$$

We can also represent edges as a complete function $f : \hat{E} \to 2^{\mathfrak{C}}$. The set E and function f are equivalent, denoted by the operator $=_\lambda$, iff the following relation holds:

$$((c, \hat{v}), (c, \hat{v}')) \in E \iff c \in f(\hat{v}, \hat{v}')$$

We write $\lambda^\emptyset$ to denote the function that maps every element to $\emptyset$, clearly $\lambda^\emptyset =_\lambda \emptyset$. We call using a set of pairs to represent vertices and edges a *set-wise* representation and using functions a *function-wise* representation.

8.2 Projections and totality

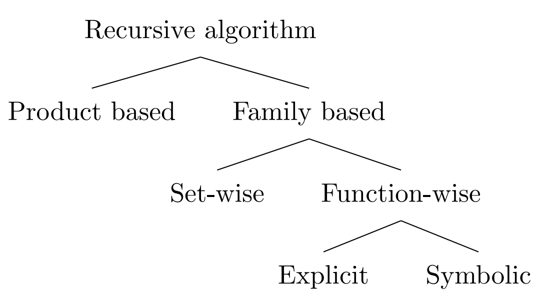
A unified PG can be projected back to one of the games from which it is the union.

Definition 8.2. *The projection of unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ to configuration c, denoted as $G_{|c}$, is the parity game $(V', \hat{V}_0, \hat{V}_1, E', \hat{\Omega})$ such that $V' = \{\hat{v} \mid (c, \hat{v}) \in V\}$ and $E' = \{(\hat{v}, \hat{w}) \mid ((c, \hat{v}), (c, \hat{w})) \in E\}$.*

One of the properties of a PG is its totality; a game is total if every vertex has at least 1 outgoing vertex. VPGs are also total, meaning that every vertex has, for every configuration $c \in \mathfrak{C}$, at least 1 outgoing vertex admitting c. Because VPGs are total their unifications are also total. Since edges in a unified PG don't cross configurations we can conclude that a unified PG is total iff every projection is total.

9 Recursive algorithm

Next we will consider Zielonka's recursive algorithm [10] which is a parity game solving algorithm that we can use to solve unified PGs. The algorithm reasons about sets of states for which certain properties hold, this makes the algorithm particularly appropriate to use on unified PGs because we can represent sets of states in unified PGs as functions that map to sets of configurations which we can represent either explicitly or symbolically. This gives rise to 4 algorithms using the recursive algorithm as its basis, we depict them in the following diagram:



9.1 Original Zielonka's recursive algorithm

First we consider the original Zielonka's recursive algorithm, created from the constructive proof given in [10], which solves total PGs. Pseudo code is presented in algorithm 1.

Algorithm 1 RECURSIVEPG($PG\ G = (V, V_0, V_1, E, \Omega)$)

$m \leftarrow \min\{\Omega(v) \mid v \in V\}$
$h \leftarrow \max\{\Omega(v) \mid v \in V\}$
if $h = m$ or $V = \emptyset$ **then**
 if h is even or $V = \emptyset$ **then**
  return $(V, \emptyset)$
 else
  return $(\emptyset, V)$
 end if
end if
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \{v \in V \mid \Omega(v) = h\}$
$A \leftarrow \alpha\text{-}Attr(G, U)$
$(W'_0, W'_1) \leftarrow$ RECURSIVEPG$(G \backslash A)$
if $W'_{\overline{\alpha}} = \emptyset$ **then**
 $W_\alpha \leftarrow A \cup W'_\alpha$
 $W_{\overline{\alpha}} \leftarrow \emptyset$
else
 $B \leftarrow \overline{\alpha}\text{-}Attr(G, W'_{\overline{\alpha}})$
 $(W''_0, W''_1) \leftarrow$ RECURSIVEPG$(G \backslash B)$
 $W_\alpha \leftarrow W''_\alpha$
 $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B$
end if
return (W_0, W_1)

An exhaustive explanation of the algorithm can be found in [10], we do introduce the definitions used in the algorithm. First we introduce the notion of an attractor set. An attractor set is a set of vertices $A \subseteq V$ calculated for player α given set $U \subseteq V$ where player α has a strategy to force the play starting in any vertex in $A \backslash U$ to a vertex in U.

Definition 9.1. *[10] Given parity game $G = (V, V_0, V_1, E, \Omega)$ and a non-empty set $U \subseteq V$ we define α-$Attr(G, U)$ such that*

$$U_0 = U$$

For $i \geq 0$:

$$\begin{aligned} U_{i+1} = U_i \cup &\{v \in V_\alpha \mid \exists v' \in V : v' \in U_i \wedge (v, v') \in E\} \\ \cup &\{v \in V_{\overline{\alpha}} \mid \forall v' \in V : (v, v') \in E \implies v' \in U_i\} \end{aligned}$$

Finally:

$$\alpha\text{-}Attr(G, U) = \bigcup_{i \geq 0} U_i$$

Next we present the definition of a subgame, where a PG and a set of vertices which are removed from the game are given.

Definition 9.2. *[10] Given a parity game $G = (V, V_0, V_1, E, \Omega)$ and $U \subseteq V$ we define the subgame $G \backslash U$ to be the game $(V', V'_0, V'_1, E', \Omega)$ with:*

- $V' = V \backslash U$,
- $V'_0 = V_0 \cap V'$,

- $V_1' = V_1 \cap V'$ *and*
- $E' = E \cap (V' \times V')$.

Note that a subgame is not necessarily total, however the recursive algorithm always creates subgames that are total (shown in [10]).

9.2 Recursive algorithm using a function-wise representation

We can modify the recursive algorithm to work with the function-wise representation of vertices and edges introduced in section 8. Pseudo code for the modified algorithm is presented in algorithm 2.

Algorithm 2 RECURSIVEUVPG($PG\ G = ($
$V : \hat{V} \to 2^{\mathfrak{C}}$,
$\hat{V}_0 \subseteq \hat{V}$,
$\hat{V}_1 \subseteq \hat{V}$,
$E : \hat{E} \to 2^{\mathfrak{C}}$,
$\hat{\Omega} : \hat{V} \to \mathbb{N}$

$m \leftarrow \min\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
$h \leftarrow \max\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
if $h = m$ or $V = \lambda^{\emptyset}$ **then**
    if h is even or $V = \lambda^{\emptyset}$ **then**
        return $(V, \lambda^{\emptyset})$
    else
        return $(\lambda^{\emptyset}, V)$
    end if
end if
$\alpha \leftarrow 0$ if h is even and 1 otherwise
$U \leftarrow \lambda^{\emptyset}$, $U(\hat{v}) \leftarrow V(\hat{v})$ for all $\hat{v}$ with $\hat{\Omega}(\hat{v}) = h$
$A \leftarrow \alpha\text{-}FAttr(G, U)$
$(W_0', W_1') \leftarrow$ RECURSIVEUVPG$(G \backslash A)$
if $W_{\overline{\alpha}}' = \lambda^{\emptyset}$ **then**
    $W_\alpha \leftarrow A \cup W_\alpha'$
    $W_{\overline{\alpha}} \leftarrow \lambda^{\emptyset}$
else
    $B \leftarrow \overline{\alpha}\text{-}FAttr(G, W_{\overline{\alpha}}')$
    $(W_0'', W_1'') \leftarrow$ RECURSIVEUVPG$(G \backslash B)$
    $W_\alpha \leftarrow W_\alpha''$
    $W_{\overline{\alpha}} \leftarrow W_{\overline{\alpha}}'' \cup B$
end if
return (W_0, W_1)

We introduce a modified attractor definition to work with the function-wise representation.

Definition 9.3. *Given unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ and a non-empty set $U \subseteq V$, both represented function-wise, we define α-FAttr(G, U) such that*

$$U_0 = U$$

For $i \geq 0$:

$$U_{i+1}(\hat{v}) = U_i(\hat{v}) \cup \begin{cases} V(\hat{v}) \cap \bigcup_{\hat{v}'} (E(\hat{v}, \hat{v}') \cap U_i(\hat{v}')) & \text{if } \hat{v} \in \hat{V}_\alpha \\ V(\hat{v}) \cap \bigcap_{\hat{v}'} ((\mathfrak{C} \backslash E(\hat{v}, \hat{v}')) \cup U_i(\hat{v}') & \text{if } \hat{v} \in \hat{V}_{\overline{\alpha}} \end{cases}$$

Finally:

$$\alpha\text{-}FAttr(G, U) = \bigcup_{i \geq 0} U_i$$

We will now prove that the function-wise attractor definition gives a result equal to the original definition.

Lemma 9.1. *Given unified PG $G = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ and set $\mathcal{U} \subseteq \mathcal{V}$ the function-wise attractor α-FAttr$(G,\mathcal{U})$ is equivalent to the set-wise attractor α-Attr$(G,\mathcal{U})$ for any $\alpha \in \{0,1\}$.*

Proof. Let V, E, U be the set-wise representation and $V^\lambda, E^\lambda, U^\lambda$ be the function-wise representation of $\mathcal{V}, \mathcal{E}, \mathcal{U}$ respectively.

The following properties hold by definition:

$$(c, \hat{v}) \in V \iff c \in V^\lambda(\hat{v})$$

$$(c, \hat{v}) \in U \iff c \in U^\lambda(\hat{v})$$

$$((c, \hat{v}), (c, \hat{v}')) \in E \iff c \in E^\lambda(\hat{v}, \hat{v}')$$

Since the attractors are inductively defined and $U_0 =_\lambda U_0^\lambda$ (because $U =_\lambda U^\lambda$) we have to prove that for some $i \geq 0$, with $U_i =_\lambda U_i^\lambda$, we have $U_{i+1} =_\lambda U_{i+1}^\lambda$, which holds iff:

$$(c, \hat{v}) \in U_{i+1} \iff c \in U_{i+1}^\lambda(\hat{v})$$

Let $(c, \hat{v}) \in V$ (and therefore $c \in V^\lambda(\hat{v})$), we consider 4 cases.

- Case: $\hat{v} \in \hat{V}_\alpha$ and $(c, \hat{v}) \in U_{i+1}$:
  To prove: $c \in U_{i+1}^\lambda(\hat{v})$.

  If $(c, \hat{v}) \in U_i$ then $c \in U_i^\lambda(\hat{v})$ and therefore $c \in U_{i+1}^\lambda(\hat{v})$. If $(c, \hat{v}) \notin U_i$ then we have $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_\alpha$ and $c \in V^\lambda(\hat{v})$ we get

  $$U_{i+1}^\lambda = \bigcup_{\hat{v}'} (E^\lambda(\hat{v}, \hat{v}') \cap U_i^\lambda(\hat{v}'))$$

  There exists an $(c', \hat{v}') \in V$ such that $(c', \hat{v}') \in U_i$ and $((c, \hat{v}), (c', \hat{v}')) \in E$. Because edges don't cross configurations we can conclude that $c' = c$. Due to equivalence we have $c \in U_i^\lambda(\hat{v}')$ and $c \in E^\lambda(\hat{v}, \hat{v}')$. If we fill this in in the above formula we can conclude that $c \in U_{i+1}^\lambda(\hat{v})$.

- Case: $\hat{v} \in \hat{V}_\alpha$ and $(c, \hat{v}) \notin U_{i+1}$:
  To prove: $c \notin U_{i+1}^\lambda(\hat{v})$.

  First we observe that since $(c, \hat{v}) \notin U_{i+1}$ we get $(c, \hat{v}) \notin U_i$ and therefore $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_\alpha$ and $c \in V^\lambda(\hat{v})$ we get

  $$U_{i+1}^\lambda = \bigcup_{\hat{v}'} (E^\lambda(\hat{v}, \hat{v}') \cap U_i^\lambda(\hat{v}'))$$

  Assume $c \in U_{i+1}^\lambda(\hat{v})$. There must exist a $\hat{v}'$ such that $c \in E^\lambda(\hat{v}, \hat{v}')$ and $c \in U_i^\lambda(\hat{v}')$. Due to equivalence we have a vertex $((c, \hat{v}), (c, \hat{v}')) \in E$ and $(c, \hat{v}') \in U_i$. In which case $(c, \hat{v})$ would be attracted and would be in U_{i+1} which is a contradiction.

- Case: $\hat{v} \in \hat{V}_{\overline{\alpha}}$ and $(c, \hat{v}) \in U_{i+1}$:
  To prove: $c \in U_{i+1}^\lambda(\hat{v})$.

  If $(c, \hat{v}) \in U_i$ then $c \in U_i^\lambda(\hat{v})$ and therefore $c \in U_{i+1}^\lambda(\hat{v})$. If $(c, \hat{v}) \notin U_i$ then we have $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we get

  $$U_{i+1}^\lambda = V^\lambda(\hat{v}) \cap \bigcap_{\hat{v}'} ((\mathfrak{C} \backslash E^\lambda(\hat{v}, \hat{v}')) \cup U_i^\lambda(\hat{v}')$$

  Assume $c \notin U_{i+1}^\lambda(\hat{v})$. Because $c \in V^\lambda(\hat{v})$ there must exist an $\hat{v}$ such that

  $$c \notin ((\mathfrak{C} \backslash E^\lambda(\hat{v}, \hat{v}')) \text{ and } c \notin U_i^\lambda(\hat{v}')$$

which is equal to
$$c \in E^\lambda(\hat{v},\hat{v}') \text{ and } c \notin U_i^\lambda(\hat{v}')$$
By equivalence we have $((c,\hat{v}),(c,\hat{v}')) \in E$ and $(c,\hat{v}') \notin U_i$. Which means that $(c,\hat{v})$ will not be attracted and $(c,\hat{v}) \notin U_{i+1}$ which is a contradiction.

- Case: $\hat{v} \in \hat{V}_{\overline{\alpha}}$ and $(c,\hat{v}) \notin U_{i+1}$:
  To prove: $c \notin U_{i+1}^\lambda(\hat{v})$.

  First we observe that since $(c,\hat{v}) \notin U_{i+1}$ we get $(c,\hat{v}) \notin U_i$ and therefore $c \notin U_i^\lambda(\hat{v})$.

  Because $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we get
  $$U_{i+1}^\lambda = V^\lambda(\hat{v}) \cap \bigcap_{\hat{v}'} ((\mathfrak{C}\backslash E^\lambda(\hat{v},\hat{v}')) \cup U_i^\lambda(\hat{v}')$$

  Since $(c,\hat{v})$ is not attracted there must exist a $(c,\hat{v}') \in V$ such that
  $$((c,\hat{v}),(c,\hat{v}')) \in E \text{ and } (c,\hat{v}') \notin U_i$$

  By equivalence we have
  $$c \in E^\lambda(\hat{v},\hat{v}') \text{ and } c \notin U_i^\lambda(\hat{v}')$$
  Which is equal to
  $$c \notin (\mathfrak{C}\backslash E^\lambda(\hat{v},\hat{v}')) \text{ and } c \notin U_i^\lambda(\hat{v}')$$
  From which we conclude
  $$c \notin ((\mathfrak{C}\backslash E^\lambda(\hat{v},\hat{v}')) \cup U_i^\lambda(\hat{v}')))$$
  Therefore we have $c \notin U_{i+1}^\lambda(\hat{v})$.

$\square$

We also introduce a modified subgame definition to work with the function-wise representation.

Definition 9.4. *For unified PG $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$, represented function-wise, and set $X \subseteq V$ we define the subgame $G\backslash X = (V', \hat{V}_0, \hat{V}_1, E', \hat{\Omega})$ such that:*

- $V'(\hat{v}) = V(\hat{v})\backslash X(\hat{v})$
- $E'(\hat{v},\hat{v}') = E(\hat{v},\hat{v}') \cap V'(\hat{v}) \cap V'(\hat{v}')$

We will now prove that this new subgame definition gives a result equal to the original subgame definition. Note that when using the original subgame definition for unified PGs we can omit the modification to the partition because, as we have seen, we can use the partitioning from the VPG in the representation of unified PGs.

Lemma 9.2. *Given unified PG $G = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ and set $\mathcal{U} \subseteq \mathcal{V}$ the subgame $G\backslash\mathcal{U} = (\mathcal{V}', \hat{V}_0, \hat{V}_1, \mathcal{E}', \hat{\Omega})$ represented set-wise is equal to the subgame represented function-wise.*

Proof. Let V, V', E, E', U be the set-wise and $V^\lambda, V^{\lambda'}, E^\lambda, E^{\lambda'}, U^\lambda$ the function-wise representations of $\mathcal{V}, \mathcal{V}', \mathcal{E}, \mathcal{E}', \mathcal{U}$ respectively. We know $V =_\lambda V^\lambda$, $E =_\lambda E^\lambda$ and $U =_\lambda U^\lambda$. To prove: $V' =_\lambda V^{\lambda'}$ and $E' =_\lambda E^{\lambda'}$.

Let $(c,\hat{v}) \in V$.

If $(c,\hat{v}) \in U$ then $c \in U^\lambda(\hat{v})$, also $(c,\hat{v}) \notin V'$ (by definition 9.2) and $c \notin V^{\lambda'}(\hat{v})$ (by definition 9.4).

If $(c,\hat{v}) \notin U$ then $c \notin U^\lambda(\hat{v})$, also $(c,\hat{v}) \in V'$ (by definition 9.2) and $c \in V^{\lambda'}(\hat{v})$ (by definition 9.4).

Let $((c,\hat{v}),(c,\hat{w})) \in E$.

If $(c,\hat{v}) \in U$ then $(c,\hat{v}) \notin V'$ and $c \notin V^{\lambda'}(\hat{v})$ (as shown above). We get $((c,\hat{v}),(c,\hat{w})) \notin V' \times V'$ so $((c,\hat{v}),(c,\hat{w})) \notin E'$ (by definition 9.2). Also $c \notin E^{\lambda'}(\hat{v},\hat{w})$ (by definition 9.4).

If $(c,\hat{w}) \in U$ then we apply the same logic.

If neither is in U then both are in V' and in $V' \times V'$ and therefore the $((c,\hat{v}),(c,\hat{w})) \in E'$. Also we get $c \in V^{\lambda'}(\hat{v})$ and $c \in V^{\lambda'}(\hat{w})$ so we get $c \in E^{\lambda'}(\hat{v},\hat{w})$ (by definition 9.4). $\square$

Next we prove the correctness of the algorithm by showing that the winning sets of the function-wise algorithm are equal to the winning sets of the set-wise algorithm.

Theorem 9.3. *Given unified PG $\mathcal{G} = (\mathcal{V}, \hat{V}_0, \hat{V}_1, \mathcal{E}, \hat{\Omega})$ the winning sets resulting from* RECURSIVEUVPG($\mathcal{G}$) *ran over the function-wise representation of $\mathcal{G}$ is equal to the winning sets resulting from* RECURSIVEPG($\mathcal{G}$) *ran over the set-wise representation of $\mathcal{G}$.*

Proof. Let $G = (V, \hat{V}_0, \hat{V}_1, E, \hat{\Omega})$ be the set-wise representation of $\mathcal{G}$ and $G^\lambda = (V^\lambda, \hat{V}_0, \hat{V}_1, E^\lambda, \hat{\Omega})$ be the function-wise representation of $\mathcal{G}$.

Proof by induction on $\mathcal{G}$.

Base When there are no vertices or only one priority RECURSIVEUVPG(G^λ) returns $\lambda^\emptyset$ and RECURSIVEPG(G) returns $\emptyset$, these two results are equal therefore the theorem holds in this case.

Step Player α gets the same value in both algorithms since the highest priority is equal for both algorithms.

Let $U = \{(c, \hat{v}) \in V \mid \hat{\Omega}(\hat{v}) = h\}$ (as calculated by RECURSIVEPG) and $U^\lambda(\hat{v}) = V^\lambda(\hat{v})$ for all $\hat{v}$ with $\hat{\Omega}(\hat{v}) = h$ (as calculated by RECURSIVEUVPG). We will show that $U =_\lambda U^\lambda$.

Let $(c, \hat{v}) \in U$ then $\hat{\Omega}(\hat{v}) = h$, therefore $U^\lambda(\hat{v}) = V^\lambda(\hat{v})$. Since $U \subseteq V$ we have $(c, \hat{v}) \in V$ and because the equality between V and V^λ we get $c \in V^\lambda(\hat{v})$ and $c \in U^\lambda(\hat{V})$.

Let $c \in U^\lambda(\hat{v})$, since $U^\lambda(\hat{v})$ is not empty we have $\hat{\Omega}(\hat{v}) = h$, furthermore $c \in V^\lambda(\hat{v})$ and therefore $(c, \hat{v}) \in V$. We can conclude that $(c, \hat{v}) \in U$ and $U =_\lambda U^\lambda$.

For the rest of the algorithm it is sufficient to see that attractor sets are equal if the game and input set are equal (as shown in lemma 9.1) and that the created subgames are equal (as shown in lemma 9.2). Since the subgames are equal we can apply the theorem on it by induction and conclude that the winning sets are also equal. □

We have seen in theorem 8.1 that solving a unified PG solves the VPG, furthermore the algorithm RECURSIVEUVPG correctly solves a unified PG therefore we can now conclude that for VPG $\hat{G}$ vertex $\hat{v}$ is won by player α for configuration c iff $c \in W_\alpha(\hat{v})$ with $(W_0, W_1) =$ RECURSIVEUVPG($G_\downarrow$).

9.2.1 Function-wise attractor set

Next we present an algorithm to calculate the function-wise attractor, the pseudo code is presented in algorithm 3. The algorithm considers vertices that are in the attracted set for some configuration, for every such vertex the algorithm tries to attract vertices that are connected by an incoming edge. If a vertex is attracted for some configuration then the incoming edges of that vertex will also be considered. We prove the correctness of the algorithm in the following lemma and theorem.

Lemma 9.4. *Vertex $\hat{v}$ and configuration c, with $c \in V(\hat{v})$, can only be attracted if there is a vertex $\hat{v}'$ such that $c \in E(\hat{v}, \hat{v}')$ and $c \in U_i(\hat{v}')$.*

Proof. We first observe that if $\hat{v} \in \hat{V}_\alpha$ then this property follows immediately from definition the function-wise attractor definition (9.3). If $\hat{v} \in \hat{V}_{\overline{\alpha}}$ we note that unified PGs are total and therefore all of their projections are also total. So vertex $\hat{v}$ has at least one outgoing edge for c, we have $\hat{w}$ such that $c \in E(\hat{v}, \hat{w})$. For $\hat{v}$ with c to be attracted we must have $c \in U_i(\hat{w})$. □

Theorem 9.5. *Set A calculated by α-*FATTRACTOR(G, U) *satisfies $A = \alpha$-FAttr(G, U).*

Proof. We will prove two loop invariants over the while loop of the algorithm.

IV1: For every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ with $c \in A(\hat{w})$ we have $c \in \alpha\text{-}FAttr(G, U)(\hat{w})$.

IV2: For every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ that can be attracted to A either $c \in A(\hat{w})$ or there exists a $\hat{w}' \in Q$ such that $c \in E(\hat{w}, \hat{w}')$.

Base: Before the loop starts we have $A = U$, therefore IV1 holds. Furthermore all the vertices that are in A for some c are also in Q so IV2 holds.

Step: Consider the beginning of an iteration and assume IV1 and IV2 hold. To prove: IV1 and IV2 hold at the end of the iteration.

Algorithm 3 α-FATTRACTOR$(G, A : \hat{V} \to 2^{\mathfrak{C}})$

1: Queue $Q \leftarrow \{\hat{v} \in \hat{V} \mid A(\hat{v}) \neq \emptyset\}$
2: **while** Q is not empty **do**
3:     $\hat{v}' \leftarrow Q.pop()$
4:     **for** $E(\hat{v}, \hat{v}') \neq \emptyset$ **do**
5:         **if** $\hat{v} \in \hat{V}_\alpha$ **then**
6:             $a \leftarrow V(\hat{v}) \cap E(\hat{v}, \hat{v}') \cap A(\hat{v}')$
7:         **else**
8:             $a \leftarrow V(\hat{v})$
9:             **for** $E(\hat{v}, \hat{v}'') \neq \emptyset$ **do**
10:                 $a \leftarrow a \cap (\mathfrak{C} \backslash E(\hat{v}, \hat{v}'') \cup A(\hat{v}''))$
11:             **end for**
12:         **end if**
13:         **if** $a \backslash A(\hat{v}) \neq \emptyset$ **then**
14:             $A(\hat{v}) \leftarrow A(\hat{v}) \cup a$
15:             $Q.push(\hat{v})$
16:         **end if**
17:     **end for**
18: **end while**
19: **return** A

Set A only contains vertices with configurations that are in α-$FAttr(G, U)$. The set is only updated through lines 5-12 and 14 of the algorithm which reflects the exact definition of the attractor set therefore IV1 holds at the end of the iteration.

Consider $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$, we distinguish three cases to prove IV2:

- $\hat{w}$ with c can be attracted by the beginning of the iteration but not by the end.

  This case can't happen because $A(\hat{w})$ only increases during the algorithm and the values for E and V are not changed throughout the algorithm.

- $\hat{w}$ with c can't be attracted by the beginning of the iteration but can by the end.

  For $\hat{w}$ with c to be able to be attracted at the end of the iteration there must be some $\hat{w}'$ with c such that during the iteration c was added to $A(\hat{w}')$ (lemma 9.4). Every $\hat{w}'$ for which $A(\hat{w}')$ is updated is added to the queue (lines 13-16). Therefore we have $\hat{w}' \in Q$ with $c \in E(\hat{w}, \hat{w}')$ and IV1 holds.

- $\hat{w}$ with c can be attracted by the beginning of the iteration and also by the end.

  Since IV2 holds at the beginning of the iteration we have either $c \in A(\hat{w})$ or we have some $\hat{w}' \in Q$ such that $c \in E(\hat{w}, \hat{w}')$. In the former case IV2 holds trivially by the end of the iteration since $A(\hat{w})$ can only increase. For the latter case we distinguish two scenario's.

  First we consider the scenario where vertex $\hat{v}'$ that is considered during the iteration (line 3 of the algorithm) is $\hat{w}'$. There is a vertex $c \in E(\hat{w}, \hat{w}')$ by IV2. Therefore we can conclude that $\hat{w}$ is considered in the for loop starting at line 4 and will be attracted in lines 5-12 and added to $A(\hat{w})$ in line 14. Therefore IV2 holds by the end of the iteration.

  Next we consider the scenario where $\hat{v}' \neq \hat{w}'$. In this case by the end of the iteration $\hat{w}'$ will still be in Q and IV2 holds.

Vertices are only added to the queue when something is added to A (if statement on line 13). This can only finitely often happen because $A(\hat{v})$ can never be larger than $V(\hat{v})$ so we can conclude that the while loop terminates after a finite number of iterations.

When the while loop terminates IV1 and IV2 hold so for every $\hat{w} \in \hat{V}$ and $c \in \mathfrak{C}$ that can be attracted to A we have $c \in A(\hat{w})$. Since we start with $A = U$ we can conclude the soundness of the algorithm. IV1 shows the completeness. $\square$

9.3 Running time

We will consider the running time for solving VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ product based and family based using the different types of representations. We will use n to denote the number of vertices, e the number of edges, c the number of configurations and d the number of distinct priorities.

The original algorithm runs in $O(e*n^d)$, if we run c parity games independently we get $O(c*e*n^d)$. We can also apply the original algorithm to a unified PG (represented set-wise) for a family based approach, in this case we get a parity game with $c*n$ vertices and $c*e$ edges. which gives a running time $O(c*e*(c*n)^d)$. However this running time can be improved by using the property that a unified PG consists of c disconnected graphs as we shown next.

We have introduced three types of family based algorithms: set-wise, function-wise with explicit configuration sets and function-wise with symbolic configuration sets. In all three algorithms the running time of the attractor set is dominant, so we need three things: analyse the running time of the base cases, analyse the running time of the attractor set and analyse the recursion.

Base cases. In the base cases the algorithm needs to do two things: find the highest and lowest priority and check if there are no more vertices in the game. For the set-wise variant we find the highest and lowest priorities by iterating all vertices, which takes $O(c*n)$. Checking if there are no more vertices is done in $O(1)$. For the function wise algorithms we can find the highest and lowest priority in $O(n)$ and checking if there are no vertices is also done in $O(n)$ since we have to check $V(\hat{v}) = \emptyset$ for every $\hat{v}$. Note that in a symbolic representation using BDDs we can check if a set is empty in $O(1)$ because the decision diagram contains a single node.

Attractor sets. For the set-wise family based approach we can use the attractor calculation from the original algorithm which has a time complexity of $O(e)$, so for a unified PG having $c*e$ edges we have $O(c*e)$.

The function-wise variants use a different attractor algorithm. First we consider the variant where sets of configurations are represented explicitly.

Consider algorithm 3. A vertex will be added to the queue when this vertex is attracted for some configuration, this can only happen $c*n$ times, once for every vertex configuration combination.

The first for loop considers all the incoming edges of a vertex. When we consider all vertices the for loop will have considered all edges, since we consider every vertex at most c times the for loop will run at most $c*e$ times in total.

The second for loop considers all outgoing edges of a vertex. The vertices that are considered are the vertices that have an edge going to the vertex being considered by the while loop. Since the while loop considers $c*n$ vertices the second for loop runs in total at most $c*n*e$ times. The loop itself performs set operations on the set of configurations which can be done in $O(c)$. This gives a total time complexity for the attractor set of $O(n*c^2*e)$.

For the symbolic representation set operations can be done in $O(c^2)$ so we get a time complexity of $O(n*c^3*e)$.

This gives the following time complexities

	Base	Attractor set
Set-wise	$O(c*n)$	$O(c*e)$
Function-wise explicit	$O(n)$	$O(n*c^2*e)$
Function-wise symbolic	$O(n)$	$O(n*c^3*e)$

Recursion. The three algorithms behave the same way with regards to their recursion, so we analyse the running time of the set-wise variant and can derive the time complexity of the others using the result.

The algorithm has two recursions, the first recursion lowers the number of distinct priorities by 1. The second recursion removes at least one edge, however the game is comprised of disjoint projections. We can use this fact use in the analyses. Consider unified PG G and A as specified by the algorithm. Now consider the projection of G to an arbitrary configuration q, $G_{|q}$. If $(G\backslash A)_{|q}$ contains a vertex that is won by player $\overline{\alpha}$ then this vertex is removed in the second recursion step. If there is no vertex won by player $\overline{\alpha}$ then the game is won in its entirety and the only vertices won by player $\overline{\alpha}$ are in different projections. We can conclude that for every configuration q the second recursion either removes a vertex or $(G\backslash A)_{|q}$ is entirely won by player α. Let $\overline{n}$ denote be the maximum number of vertices that are won by player $\overline{\alpha}$ in game $(G\backslash A)_{|q}$. Since every projection has at most n vertices the value for $\overline{n}$ can be at most n. Furthermore since $\overline{n}$ depends on A, which depends on the maximum priority, the value $\overline{n}$ gets reset when the top priority is removed in the first recursion. We can now write down the recursion of the algorithm:

$$T(d, \overline{n}) \leq T(d-1, n) + T(d, \overline{n}-1) + O(c * e)$$

When $\overline{n} = 0$ we will get $W_{\overline{\alpha}} = \emptyset$ as a result of the first recursion. In such a case there will be only 1 recursion.

$$T(d, 0) \leq T(d-1, n) + O(c * e)$$

Finally we have the base case where there is 1 priority:

$$T(1, \overline{n}) \leq O(c * n)$$

Expanding the second recursion gives

$$\begin{aligned} T(d) &\leq (n+1)T(d-1) + (n+1)O(c * e) \\ T(1) &\leq O(c * n) \end{aligned}$$

We will now prove that $T(d) \leq (n+d)^d O(c * e)$ by induction on d.
Base $d = 1$: $T(1) \leq O(c * n) \leq O(c * e) \leq (n+1)^1 O(c * e)$
Step $d > 1$:

$$\begin{aligned} T(d) &\leq (n+1)T(d-1) + (n+1)O(c * e) \\ &\leq (n+1)(n+d-1)^{d-1} O(c * e) + (n+1)O(c * e) \end{aligned}$$

Since $n + 1 \leq n + d - 1$ we get:

$$\begin{aligned} T(d) &\leq (n+d-1)(n+d-1)^{d-1} O(c * e) + (n+1)O(c * e) \\ &\leq (n+d-1)^d O(c * e) + (n+1)O(c * e) \\ &\leq ((n+d-1)^d + n + 1)O(c * e) \end{aligned}$$

Using lemma A.1 we get

$$T(d) \leq (n+d)^d O(c * e)$$

This gives a time complexity of $O(c * e * (n+d)^d) = O(c * e * n^d)$. Note that the base time complexity is subsumed in the recursion by the time complexity of the attractor set. Since the time complexity of the attractor set is higher that the time complexity of the base cases for all three variants of algorithms we can simply fill in the attractor time complexity to get $O(n * c^2 * e * n^d)$ for the function-wise explicit algorithm and $O(n * c^3 * e * n^d)$ for the function-wise symbolic algorithm.

The different algorithms, including their time complexities, are repeated in the diagram below:

- Recursive algorithm
  - Product based $O(c * e * n^d)$
  - Family based
    - Set-wise $O(c * e * n^d)$
    - Function-wise
      - Explicit $O(n * c^2 * e * n^d)$
      - Symbolic $O(n * c^3 * e * n^d)$

The function wise time complexities consist of three parts:

- the number of edges in the queue during the attractor calculation,
- the time complexity of set operations on subsets of $\mathfrak{C}$ and
- the number of recursions.

The number of vertices in the queue during attracting is at most $c * n$, however this number will only be large if we attract a very small number of configurations at a per time we evaluate an edge. Most likely we will be able to attract many configurations at the same time, especially when the VPG originates from an FTS there is a good change that many edges admit most or all configurations in $\mathfrak{C}$. So when there are many similarities in behaviour between the different configurations in the FTS we will have a low number of vertices in the queue.

The time complexity of set operations is $O(c)$ when using an explicit representation and $O(c^2)$ when using a symbolic one. However, as shown in [9], a breadth-depth first implementation of BDDs keeps a table of already computed results. This allows us to get already calculated results in sublinear time. In total there are 2^c possible sets and therefore 2^{2c} possible set combinations and $O(2^c)$ possible set operations that can be computed. However when solving a VPG originating from an FTS there will most likely be a relatively small number of different edge guards, in which case the number of unique sets considered in the algorithm will be small and we can often retrieve a set calculation from the computed table.

We can see that even though the running time of the family based symbolic algorithm is the worse, its actual running time might be good when we are able to attract multiple configurations at the same time and have a small number of different edge guards.

10 Pessimistic parity games

Given a VPG with configurations $\mathfrak{C}$ we can try to determine sets P_0, P_1 such that the vertices in set P_α are won by player $\alpha \in \{0, 1\}$ for any configuration in $\mathfrak{C}$. We can do so by creating a *pessimistic* PG; a pessimistic PG is a parity game created from a VPG for a player $\alpha \in \{0, 1\}$ such that the PG allows all edges that player $\overline{\alpha}$ might take but only allows edges for α when that edge admits all the configurations in $\mathfrak{C}$.

Definition 10.1. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$, we can create pessimistic PG $G_{\triangleright\alpha}$ for player $\alpha \in \{0, 1\}$. We have*

$$G_{\triangleright\alpha} = \{V, V_0, V_1, E', \Omega\}$$

such that

$$E' = \{(v, w) \in E \mid v \in V_{\overline{\alpha}} \vee \theta(v, w) = \mathfrak{C}\}$$

Note that pessimistic parity games are not necessarily total. A play in a PG that is not total might result in a finite path, in such a case the player that can't make a move looses the play.

When solving a pessimistic PG $G_{\triangleright\alpha}$ we get winning sets W_0, W_1, every vertex in W_α is winning for player α in G played for any configuration, as shown in the following theorem.

Theorem 10.1. *Given:*

- *VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$,*
- *configuration $c \in \mathfrak{C}$,*
- *winning sets W_0^c, W_1^c for game G,*
- *player $\alpha \in \{0, 1\}$ and*
- *pessimistic PG $G_{\triangleright\alpha}$ with winning sets P_0 and P_1*

we have $P_\alpha \subseteq W_\alpha^c$.

Proof. Player α has a strategy in game $G_{\triangleright\alpha}$ such that vertices in P_α are won. We will show that this strategy can also be applied to game $G_{|c}$ to win the same or more vertices.

First we observe that any edge that is taken by player α in game $G_{\triangleright\alpha}$ can also be taken in game $G_{|c}$ so player α can play the same strategy in game $G_{|c}$.

For player $\overline{\alpha}$ there are possibly edges that can be taken in $G_{\triangleright\alpha}$ but can't be taken in $G_{|c}$, in such a case player $\overline{\alpha}$'s choices are limited in game $G_{|c}$ compared to $G_{\triangleright\alpha}$ so if player $\overline{\alpha}$ can't win a vertex in $G_{\triangleright\alpha}$ then he/she can't win that vertex in $G_{|c}$.

We can conclude that applying the strategy from game $G_{\triangleright\alpha}$ in game $G_{|c}$ for player α wins the same or more vertices. □

10.1 Configuration partitioning

Definition 10.2. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ and non-empty set $\mathfrak{X} \subseteq \mathfrak{C}$ we define the subgame $G \cap \mathfrak{X} = (V, V_0, V_1, E', \Omega, \mathfrak{C}', \theta')$ such that*

- $\mathfrak{C}' = \mathfrak{C} \cap \mathfrak{X}$,
- $\theta'(e) = \theta(e) \cap \mathfrak{C}'$ *and*
- $E' = \{e \in E \mid \theta'(e) \neq \emptyset\}$.

VPGs are total, meaning that for every configuration and every vertex there is an outgoing edge from that vertex admitting that configuration. In subgames the set of configurations is restricted and only edge guards and edges are removed for configurations that fall outside the restricted set, therefore we still have totality.

Furthermore it is trivial to see that every projection $G_{|c}$ is equal to $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$.

Finally the subset operator is associative, meaning $(G \cap \mathfrak{X}) \cap \mathfrak{X}' = G \cap (\mathfrak{X} \cap \mathfrak{X}') = G \cap \mathfrak{X} \cap \mathfrak{X}'$.

Vertices in winning set P_α for $G_{\triangleright\alpha}$ are also winning for player α in pessimistic subgames of G, as shown in the following lemma.

Lemma 10.2. *Given:*

- *VPG* $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$,
- P_0 *being the winning set of game* $G_{\triangleright 0}$ *for player* 0,
- P_1 *being the winning set of game* $G_{\triangleright 1}$ *for player* 1,
- *non-empty set* $\mathfrak{X} \subseteq \mathfrak{C}$,
- *player* $\alpha \in \{0, 1\}$ *and*
- *winning sets* Q_0, Q_1 *for game* $(G \cap \mathfrak{X})_{\triangleright\alpha}$

we have

$$P_0 \subseteq Q_0$$
$$P_1 \subseteq Q_1$$

Proof. Let edge (v, w) be an edge in game $G_{\triangleright\alpha}$ with $v \in V_\alpha$. Edge (v, w) admits all configuration in $\mathfrak{C}$ so it also admits all configuration in $\mathfrak{C} \cap \mathfrak{X}$, therefore we can conclude that edge (v, w) is also an edge of game $(G \cap \mathfrak{X})_{\triangleright\alpha}$.

Let edge (v, w) be an edge in game $(G \cap \mathfrak{X})_{\triangleright\alpha}$ with $v \in V_{\overline{\alpha}}$. The edge admits some configuration in $\mathfrak{C} \cap \mathfrak{X}$, this configuration is also in $\mathfrak{C}$ so we can conclude that edge (v, w) is also an edge of game $G_{\triangleright\alpha}$.

We have concluded that game $(G \cap \mathfrak{X})_{\triangleright\alpha}$ has the same or more edges for player α as game $G_{\triangleright\alpha}$ and the same or less edges for player $\overline{\alpha}$. Therefore we can conclude that any vertex won by player α in $G_{\triangleright\alpha}$ is also won by α in game $(G \cap \mathfrak{X})_{\triangleright\alpha}$, ie. $P_\alpha \subseteq Q_\alpha$.

Let $v \in P_{\overline{\alpha}}$, using theorem 10.1 we find that v is winning for player $\overline{\alpha}$ in $G_{|c}$ for any $c \in \mathfrak{C}$. Because projections of subgames are the same as projections of the original game we can conclude that v is winning for player $\overline{\alpha}$ in $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$.

Assume $v \notin Q_{\overline{\alpha}}$ then $v \in Q_\alpha$ and using theorem 10.1 we find that v is winning for player α in $(G \cap \mathfrak{X})_{|c}$ for any $c \in \mathfrak{C} \cap \mathfrak{X}$. This is a contradiction so we can conclude $v \in Q_{\overline{\alpha}}$ and therefore $P_{\overline{\alpha}} \subseteq Q_{\overline{\alpha}}$. □

11 Incremental pre-solve algorithm

Using pessimistic games we can create an algorithm that solves the entire VPG incrementally. First we try to find P_0 and P_1 for all the configurations, next we partition the configuration set in two sets and try to improve P_0 and P_1 for these sets. We continue to do this until we have configuration sets of size 1 and we simply solve the projection.

The pseudo code is presented in algorithm 4, the algorithm relies on a SOLVE algorithm that solves a parity game. The SOLVE algorithm is some algorithm that solves parity games and can use the parameters P_0 and P_1 to more efficiently solve the parity game. The pessimistic parity games are not necessarily total so the SOLVE algorithm must be able to solve non-total games. When solving a parity game for which we have P_0 and P_1 we say that vertices in P_0 and P_1 are *pre-solved*.

Algorithm 4 INCPRESOLVE$(G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta), P_0, P_1)$

1: **if** $|\mathfrak{C}| = 1$ **then**
2:     $\{c\} \leftarrow \mathfrak{C}$
3:     $(W_0', W_1') \leftarrow \text{SOLVE}(G_{|c}, P_0, P_1)$
4:     **return** $(\mathfrak{C} \times W_0', \mathfrak{C} \times W_1')$
5: **end if**
6: $(P_0', -) \leftarrow \text{SOLVE}(G_{\triangleright 0}, P_0, P_1)$
7: $(-, P_1') \leftarrow \text{SOLVE}(G_{\triangleright 1}, P_0, P_1)$
8: **if** $P_0' \cup P_1' = V$ **then**
9:     **return** $(\mathfrak{C} \times P_0', \mathfrak{C} \times P_1')$
10: **end if**
11: $\mathfrak{C}^a, \mathfrak{C}^b \leftarrow$ partition $\mathfrak{C}$ in non-empty parts
12: $(W_0^a, W_1^a) \leftarrow \text{INCPRESOLVE}(G \cap \mathfrak{C}^a, P_0', P_1')$
13: $(W_0^b, W_1^b) \leftarrow \text{INCPRESOLVE}(G \cap \mathfrak{C}^b, P_0', P_1')$
14: $W_0 \leftarrow W_0^a \cup W_0^b$
15: $W_1 \leftarrow W_1^a \cup W_1^b$
16: **return** (W_0, W_1)

A SOLVE algorithm must correctly solve a game as long as the sets P_0 and P_1 are in fact vertices that are won by player 0 and 1 respectively. We prove that this is the case in the INCPRESOLVE algorithm.

Theorem 11.1. *Given VPG $\hat{G}$. For every* SOLVE(G, P_0, P_1) *that is invoked during* INCPRESOLVE$(\hat{G}, \emptyset, \emptyset)$ *we have winning sets W_0, W_1 for game G for which the following holds:*

$$P_0 \subseteq W_0$$

$$P_1 \subseteq W_1$$

Proof. When $P_0 = \emptyset$ and $P_1 = \emptyset$ the theorem holds trivially. So we will start the analyses after the first recursion.

After the first recursion the game is $\hat{G} \cap \mathfrak{X}$ with $\mathfrak{X}$ being either $\mathfrak{C}^a$ or $\mathfrak{C}^b$. The set P_0 is the winning set for player 0 for game $\hat{G}_{\triangleright 0}$ and the set P_1 is the winning set for player 1 for game $\hat{G}_{\triangleright 1}$. In the next recursion the game is $\hat{G} \cap \mathfrak{X} \cap \mathfrak{X}'$ with P_0 being the winning set for player 0 in game $(\hat{G} \cap \mathfrak{X})_{\triangleright 0}$ and P_1 being the winning set for player 1 in game $(\hat{G} \cap \mathfrak{X})_{\triangleright 1}$. After the first recursion the game is always of the form $(\hat{G} \cap \mathfrak{X}^0 \cap \dots \cap \mathfrak{X}^{k-1}) \cap \mathfrak{X}^k$. Furthermore P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \dots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \dots \cap \mathfrak{X}^{k-1})_{\triangleright 1}$.

Next we inspect the three places SOLVE is invoked:

1. Consider the case where there is only one configuration in $\mathfrak{C}$ (line 1-5). Because P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \dots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ the vertices in P_0 are won by player 0 in game $G_{|c}$ for all $c \in \mathfrak{X}^0 \cap \dots \cap \mathfrak{X}^{k-1}$ (using theorem 10.1). This includes the one element in $\mathfrak{C}$. So we can conclude $P_0 \subseteq W_0$ where W_0 is the winning set for player 0 in game $G_{|c}$ where $\{c\} = \mathfrak{C}$.

   Similarly for player 1 we can conclude $P_1 \subseteq W_1$ and the theorem holds in this case.

2. On line 6 the game $G_{\triangleright 0}$ is solved with P_0 and P_1. Because $G = \hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1} \cap \mathfrak{X}^k$ and P_0 is the winning set for player 0 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 0}$ and P_1 is the winning set for player 1 for game $(\hat{G} \cap \mathfrak{X}^0 \cap \cdots \cap \mathfrak{X}^{k-1})_{\triangleright 1}$ we can apply lemma 10.2 to conclude that the theorem holds in this case.

3. On line 7 we apply the same reasoning and lemma to conclude that the theorem holds in this case.

□

Next we prove the correctness of the algorithm, assuming the correctness of the SOLVE algorithm.

Theorem 11.2. *Given VPG* $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$ *and* $(W_0, W_1) = \text{INCPRESOLVE}(\hat{G}, \emptyset, \emptyset)$. *For every configuration* $c \in \mathfrak{C}$ *and winning sets* $\hat{W}_0^c, \hat{W}_1^c$ *for game* $\hat{G}$ *player for* c *it holds that:*

$$(c, v) \in W_0 \iff v \in \hat{W}_0^c$$

$$(c, v) \in W_1 \iff v \in \hat{W}_1^c$$

Proof. We will prove the theorem by applying induction on $\mathfrak{C}$.

Base $|\mathfrak{C}| = 1$, when there is only one configuration, being c, then the algorithm solves game $G_{|c}$. The product of the winning sets and $\{c\}$ is returned, so the theorem holds.

Step Consider P_0' and P_1' as calculated in the algorithm (line 6-7). By theorem 10.1 all vertices in P_0' are won by player 0 in game $G_{|c}$ for any $c \in \mathfrak{C}$, similarly for P_1' and player 1.

If $P_0' \cup P_1' = V$ then the algorithm returns $(\mathfrak{C} \times P_0', \mathfrak{C} \times P_1')$. In which case the theorem holds because there are no configuration vertex combinations that are not in either winning set and theorem 10.1 proves the correctness.

If $P_0' \cup P_1' \neq V$ then we have winning sets (W_0^a, W_1^a) for which the theorem holds (by induction) for game $G \cap \mathfrak{C}^a$ and (W_0^b, W_1^b) for which the theorem holds (by induction) for game $G \cap \mathfrak{C}^b$. The algorithm returns $(W_0^a \cup W_0^b, W_1^a \cup W_1^b)$. Since $\mathfrak{C}^a \cup \mathfrak{C}^b = \mathfrak{C}$ all vertex configuration combinations are in the winning sets and the correctness follows from induction. □

11.1 Fixed-point approximation algorithm

Parity games can be solved by solving an alternating fixed point formula, as shown in [11]. We will consider PG $G = (V, V_0, V_1, E, \Omega)$ with d distinct priorities. We can apply *priority compression* to make sure every priority in G maps to a value in $\{0, \ldots, d-1\}$ or $\{1, \ldots, d\}$ [12, 13]. We assume without loss of generality that the priorities map to $\{0, \ldots, d-1\}$ and that $d-1$ is even.

Consider the following formula

$$S(G = (V, V_0, V_1, E, \Omega)) = \nu Z_{d-1}.\mu Z_{d-2}.\ldots.\nu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$$

with

$$F_0(Z_{d-1}, \ldots, Z_0) = \{v \in V_0 \mid \exists_{w \in V}(v, w) \in E \wedge Z_{\Omega(w)}\} \cup \{v \in V_1 \mid \forall_{w \in V}(v, w) \in E \implies Z_{\Omega(w)}\}$$

where $Z_i \subseteq V$. The formula $\nu X.f(X)$ solves the greatest fixed-point of X in f, similarly $\mu X.f(X)$ solves the least fixed-point of X in f.

To understand the formula we consider sub-formula $\nu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$. This formula holds for vertices from which player 0 can either force the play into a node with priority $i > 0$ for which Z_i holds or the player can stay in vertices with priority 0 indefinitely. The formula $\mu Z_0.F_0(Z_{d-1}, \ldots, Z_0)$ holds for vertices from which player 0 can force the play into a node with priority $i > 0$ for which Z_i holds in finitely many steps.

As shown in [11], solving $S(G)$ gives the winning set for player 0 in game G. A concrete algorithm is introduced in [13], note that this algorithm can solve finite games. We will extend this algorithm such that it calculates $S(G)$ in an efficient manner by using P_0 and P_1 where it is known beforehand that vertices in P_0 are won by player 0 and vertices in P_1 are won by player 1.

11.1.1 Fixed-point approximation

As shown in [16] we can calculate fixed-point $\mu X.f(X)$ when f is monotonic in X by approximating X.

$$\mu X.f(X) = \bigcup_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \subseteq \mu X.f(X)$. So picking the smallest value possible for X_0 will always correctly calculate $\mu X.f(X)$.

Similarly we can calculate fixed-point $\nu X.f(X)$ when f is monotonic in X by approximating X.

$$\nu X.f(X) = \bigcap_{i \geq 0} X^i$$

where $X^i = f(X^{i-1})$ for $i > 0$ and $X^0 \supseteq \nu X.f(X)$. So picking the largest value possible for X_0 will always correctly calculate $\nu X.f(X)$.

Clearly the formula $F_0(Z_{d-1}, \ldots, Z_0)$ is monotonic in any Z_j, so we can calculate Z_{d-1} by approximating every Z_j starting at V for νZ_j and starting at $\emptyset$ for μZ_j.

11.1.2 Pre-solved games

Let G be a PG and let sets P_0 and P_1 be such that vertices in P_0 are won by player 0 and vertices in P_1 are won by player 1. We can fixed-point approximate $S(G)$ to calculate W_0, we know that W_0 is bounded by P_0 and P_1, specifically we have

$$P_0 \subseteq W_0 \subseteq V \backslash P_1$$

We can use this restriction to efficiently approximate $S(G)$. If no bounds are know and we would approximate fixed-point formula $\nu Z_{d-1} \ldots$ then we would start at $Z^0_{d-1} = V$ which is the largest value possible, however given the bounds we can start our approximations of greatest fixed-point variables at $V \backslash P_1$ and start our approximations of least fixed-point variables at P_0. The following lemma's and theorems prove this.

Lemma 11.3. *Given*

- *A complete lattice* $\langle 2^A, \subseteq \rangle$,
- *monotonic function* $f : 2^A \to 2^A$ *and*
- $R^\perp \subseteq A$ *and* $R^\top \subseteq A$ *such that* $R^\perp \subseteq \nu X.f(X) \subseteq R^\top$

we approximate X *by starting with* $X^0 = R^\top$. *For any* $i \geq 0$ *it holds that*

$$R^\perp \subseteq f(X^i) \subseteq R^\top$$

Proof. Assume $R^\perp \supset f(X^i)$. By fixed-point approximation we have $\nu X.f(X) = \cap_{j \geq 0} X^j$, so we find $R^\perp \supset \nu X.f(x)$ which is a contradiction so $R^\perp \subseteq f(X^i)$.

Assume $f(X^i) \supset R^\top$. Because of monotonicity we find $X^i \subseteq R^\top$ and therefore $f(X^i) \supset R^\top \supseteq X^i$. Using the Knaster-Tarski theorem (7.1) we can conclude that the greatest fixed-point of $f(X)$ is larger than $f(X^i)$, so we find $\nu X.f(X) \supset R^\top$ which is a contradiction so $f(X^i) \subseteq R^\top$. □

Lemma 11.4. *Given*

- *A complete lattice* $\langle 2^A, \subseteq \rangle$,
- *monotonic function* $f : 2^A \to 2^A$ *and*
- $R^\perp \subseteq A$ *and* $R^\top \subseteq A$ *such that* $R^\perp \subseteq \mu X.f(X) \subseteq R^\top$

we approximate X *by starting with* $X^0 = R^\perp$. *For any* $i \geq 0$ *it holds that*

$$R^\perp \subseteq f(X^i) \subseteq R^\top$$

Theorem 11.5. *Given PG $G = (V, V_0, V_1, E, \Omega)$ with P_0 and P_1 such that vertices in P_0 are won by player 0 in game G and vertices in P_1 are won by player 1 in game G we can approximate the fixed-point variables by starting at P_0 for least fixed-points and starting at $V \backslash P_1$ for greatest fixed-points.*

Proof. Let $f(Z_{d-1}) = \mu Z_{d-2} \dots \nu Z_0.F_0(Z_{d-1}, \dots, Z_0)$. Because $\nu Z_{d-1}.f(Z_{d-1})$ calculates W_0 we know $P_0 \subseteq \nu Z_{d-1}.f(Z_{d-1}) \subseteq V \backslash P_1$ so we can start the fixed-point approximation at $Z^0_{d-1} = V \backslash P_1$. Using lemma 11.3 we find for any $i \geq 0$ we have $P_0 \subseteq f(Z^i_{d-1}) \subseteq V \backslash P_1$.

Let $g(Z_{d-2}) = \nu Z_{d-3} \dots \nu Z_0.F_0(Z^i_{d-1}, Z_{d-2}, \dots, Z_0)$. We found $P_0 \subseteq f(Z^i_{d-1}) \subseteq V \backslash P_1$, therefore we have $P_0 \subseteq \mu Z_{d-2}.g(Z_{d-2}) \subseteq V \backslash P_1$ so we can start the fixed-point approximation at $Z^0_{d-2} = P_0$. Using lemma 11.4 we find that for any $j \geq 0$ we have $P_0 \subseteq g(Z^j_{d-2}) \subseteq V \backslash P_1$.

We can repeat this logic up until Z_0 to conclude that the theorem holds. □

We can now take the fixed-point algorithm presented in [13] and modify it by starting at P_0 and $V \backslash P_1$. The pseudo code is presented in algorithm 5, its correctness follows from [13] and theorem 11.5.

Algorithm 5 Fixed-point iteration with P_0 and P_1

function FPITER($G = (V, V_0, V_1, E, \Omega), P_0, P_1$)
  for $i \leftarrow d-1, \dots, 0$ **do**
    INIT(i)
  end for
  repeat
    $Z'_0 \leftarrow Z_0$
    $Z_0 \leftarrow$ DIAMOND() $\cup$ BOX()
    $i \leftarrow 0$
    while $Z_i = Z'_i \wedge i < d-1$ **do**
      $i \leftarrow i+1$
      $Z'_i \leftarrow Z_i$
      $Z_i \leftarrow Z_{i-1}$
      INIT($i-1$)
    end while
  until $i = d-1 \wedge Z_{d-1} = Z'_{d-1}$
  return $(Z_{d-1}, V \backslash Z_{d-1})$
end function

function INIT(i)
  $Z_i \leftarrow P_0$ if i is odd, $V \backslash P_1$ otherwise
end function

function DIAMOND
  return $\{v \in V_0 \mid \exists_{w \in V} (v, w) \in E \wedge w \in Z_{\Omega(w)}\}$
end function

function BOX
  return $\{v \in V_1 \mid \forall_{w \in V} (v, w) \in E \implies w \in Z_{\Omega(w)}\}$
end function

This algorithm is appropriate to use as a SOLVE algorithm in the INCPRESOLVE since it solves parity games that are not necessarily total and uses P_0 and P_1.

11.2 Running time

We will consider the running time for solving VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ product based and family based. We will use n to denote the number of vertices, e the number of edges, c the number of configurations and d the number of distinct priorities.

The fixed-point iteration algorithm without P_0 and P_1 runs in $O(e * n^d)$ ([13]). We can use this algorithm to solve G product based, ie. solve all the projections of G. This gives a time complexity of $O(c * e * n^d)$.

Next we consider the INCPRESOLVE algorithm for a family based approach, observe that in the worst case we have to split the set of configurations all the way down to individual configurations. We can consider the recursion as a tree where the leafs are individual configurations and at every internal node the set of configurations is split in two. Since in the worst case there are c leaves, there are at most $c-1$ internal nodes. At every internal node the algorithm solves two games and at every leaf the algorithm solves 1 game, so we get $c + 2c - 2 = O(c)$ games that are being solved by INCPRESOLVE. In the worst case there are no similarities between the configuration in G and at every iteration P_0 and P_1 are empty. In this case the FPITE algorithm behaves the same as the original algorithm and has a time complexity of $O(e * n^d)$, this gives an overall time complexity of $O(c * e * n^d)$ which is equal to a product based approach.

Part III

Experimental evaluation

12 Introduction

The algorithms proposed to solve VPGs don't have a better worst case running time complexity, generally speaking this is the case because the projections of the VPG might have very little in common. However the aim of the algorithms is to solve VPGs effectively when there are commonalities. In order to evaluate the performance of the algorithms when there are commonalities the algorithms are implemented and ran on a different VPGs.

13 Implementation

The algorithms are implemented in C++ version 14.

13.1 BDDs

BuDDy[1] as a BDD library. Cache is reset after parsing.

13.2 Recursive algorithm

Three variants of the recursive algorithm are implemented:

- The original algorithm that solves parity games.
- The functions-wise variant using explicit configuration representation.
- The function-wise variant using symbolic configuration representation.

13.3 Incremental pre-solve algorithm

13.3.1 Fixed-point iteration algorithm

The fixed-point iteration algorithm is implemented to be applied to parity games or to be used by the assistance finding algorithm where it will start the fixed-point variables at some value passed on by the assistance finding algorithm.

Fixed-point variables are bitvectors representing a subset of vertices. Applied the three optimizations described in [13]:

- For fixed-point variable Z_i its value is only ever used to check if a vertex with priority i is in Z_i. So instead of storing all vertices in Z_i we only have to store the vertices that have priority i. We can store all fixed-point variables in a single bitvector, named Z, of size n.
- The algorithm only reinitializes a certain range of fixed-point variables. So the diamond and box operations can use the previous result and only reconsider vertices that have an edge to a vertex that has a priority for which its fixed-point variable is reset.
- The algorithm updates variables Z_0 to Z_m and reinitializes Z_0 to Z_{m-1}, however if Z_m is a least fixed-point variable then Z_m has just increased and due to monotonicity the other least fixed-point formula's, ie. $Z_{m-2}, Z_{m-4}, \ldots$, will also increase so there is no need to reset them. Similarly for greatest fixed-point variables. So we only to reset half of the variables instead of all of them.

[1] https://sourceforge.net/projects/buddy/

Finally the vertices in the game are reordered such that they are ordered by parity first and secondly by priority. Using the above optimizations the algorithm needs to reset variables $Z_m, Z_{m-2}, \ldots$, these variables are stored in a single bitvector Z. By reordering the variables to be sorted by parity and priority these vertices that need to be reset are always consecutively stored in Z, so resetting this sequence can be done by a memory copy instead of iterating all the different vertices. Note that when the algorithm is used by the assistance finding algorithm the variables are not reset to simply $\emptyset$ and V but are reset to two specific bitvectors that are given by the assistance finding algorithm. These bitvectors have the same order and resetting can be done by copying a part of them into Z.

14 Test cases

minepump

elevator

14.1 Random games

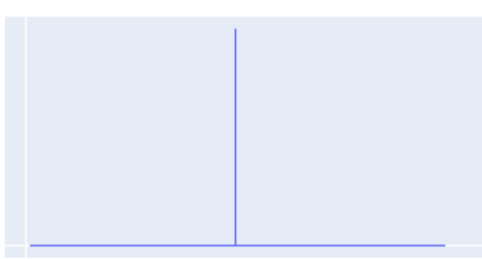
We can create a random variability parity game by creating a random parity game and creating sets of configurations that guard the edges. For these sets we need to consider two factors: how large are the sets guarding the edges and how are they created.

The guard sets in the minepump and elevator games have a very specific distribution where on average 99% of the sets admit either 100% or 50% of the configurations. An edge requiring the presence or absence of a specific feature results in a set admitting 50%, this explains the distribution. The average edge in the examples admits 92% of the configurations.

We will use λ to denote the average size of guard sets in VPGs. We will create random games with a specific λ, we do this by using a probabilistic distribution ranging from 0 to 1 to determine the size of every guard set. Such a distribution will have a mean equal to λ. We will consider two distributions:

- A modified Bernoulli distribution; in a Bernoulli distribution there is a probability of p to get an outcome of 1 and a probability of $1-p$ to get an outcome of 0. We modify this such that there is a probability of p to get 1 and a probability of $1-p$ to get 0.5. This gives a mean of $1p + 0.5(1-p) = 0.5p + 0.5$. So to get a mean of λ we choose $p = 2\lambda - 1$. Note that we can't use this distribution when $\lambda < 0.5$ because p becomes larger than 1.
- A beta distribution; a beta distribution ranges from 0 to 1 and is curved such that it has a specific mean. The beta distribution has two parameters: α and β and a mean of $\frac{\alpha}{\alpha+\beta}$. We will pick $\beta = 1$ and $\alpha = \frac{\lambda\beta}{1-\lambda}$ to get a mean of λ.

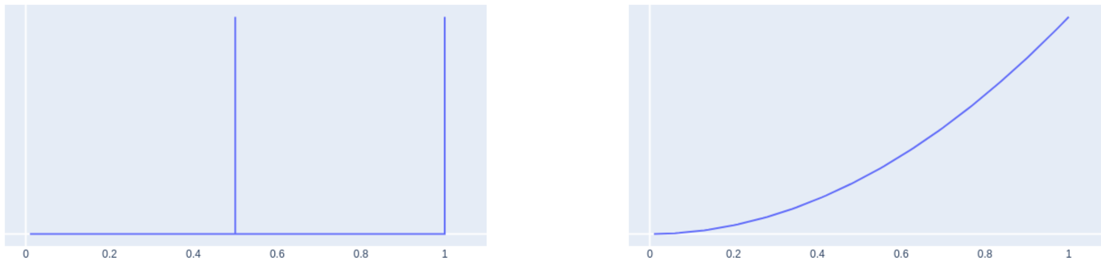
Figures 12, 13 and 14 show the shapes of the distribution for different values for λ.



(a) Modified Bernoulli distribution with $p = 0$

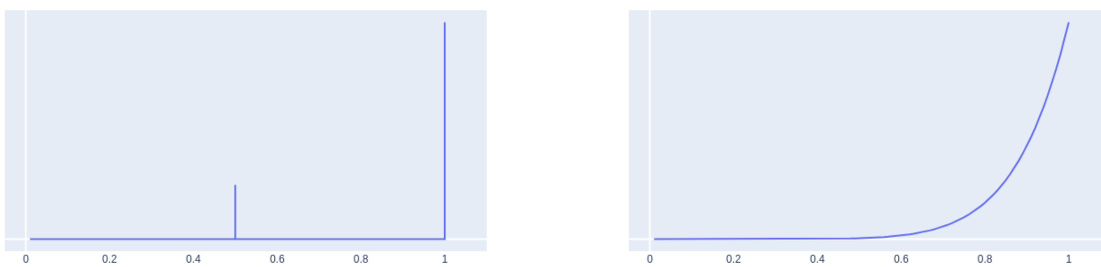
(b) Beta distribution with $\beta = 1$ and $\alpha = 1$

Figure 12: Edge guard size distribution for $\lambda = 0.5$



(a) Modified Bernoulli distribution with $p = 0.5$ (b) Beta distribution with $\beta = 1$ and $\alpha = 3$

Figure 13: Edge guard size distribution for $\lambda = 0.75$



(a) Modified Bernoulli distribution with $p = 0.8$ (b) Beta distribution with $\beta = 1$ and $\alpha = 9$

Figure 14: Edge guard size distribution for $\lambda = 0.9$

Next we need to consider how the set of configuration is created. We can simply create a random set of configurations without any notion of features, we call this a *configuration based* approach. Alternatively we can use a *feature based* approach where we create sets by looking at features. Consider features $f_0, \ldots, f_m$, we can create a boolean function that is the disjunction of k features where every feature in the disjunction has probability 0.5 of being negated. For example when using $k = 3$ and $m = 5$ we might get boolean formula $f_1 \vee \neg f_2 \vee \neg f_4$. Such a boolean formula corresponds to a set of configurations of size 2^{m-k} and a relative size $\frac{2^{m-k}}{2^m} = 2^{-k}$, so when creating a set with relative size r we choose $k = \min(m, \lfloor -\log_2 r \rfloor)$. When using a feature based approach we can only create sets that have a relative size of $\frac{1}{2^i}$ for some $i \in \mathbb{N}$.

We can create 4 types of games:

1. Bernoulli distributed and feature based. These games are most similar to the model verification games.

2. Bernoulli distributed and configuration based. These games do have the characteristics of a model verification game in terms of set size but have unstructured sets guarding the edges. Furthermore with a configuration based approach less guard sets will be identical than with a feature based approach.

3. Beta distributed and configuration based. These games are most different from the model verification games.

4. Beta distributed and feature based. Games created in this way don't have the average relative set size of λ because the feature based approach can only create sets of size $\frac{1}{2^i}$ for any $\lambda \geq \frac{1}{2}$ all the sets have

either relative size $\frac{1}{2}$ or 1, so effectively this creates the same games as using the Bernoulli distribution only with an incorrect average relative set size. Therefore we will not consider this category of games.

14.2 Categories

For random games of type 1,2 and 3 we create 50 games: game 50 to game 99, where game i has $\lambda = \frac{i}{100}$ and a random number of features, nodes, edges and maximum priority.

Furthermore we create 48 games to evaluate how the algorithm scales when the number of features becomes larger. For every $i \in [2, 12]$ we create a random games ia, ib, ic and id of type 1 with $\lambda = 0.92$, i features and a random number of nodes, edges and maximum priority.

15 Results

In this section the experimental results are presented. The 5 categories of problems are ran against the following algorithms:

1. Zielonka's recursive algorithm, product based
2. Fixed-point iteration, product based
3. Fixed-point iteration, local product based
4. Zielonka's recursive algorithm, family based with explicit configuration representation
5. Zielonka's recursive algorithm, family based with symbolic configuration representation
6. Incremental pre-solve algorithm
7. Incremental pre-solve algorithm, local

The results presented is the time it took to solve the games, parsing times and projection (for product based approaches) are excluded. So the product based results are the sum of the solve times of the projections, parsing and projecting are not included in the result.

We use the Zielonka's product based algorithm as the algorithm to compare the family based approaches against. This is the most widely used algorithm and in most cases outperforms the fixed-point iteration algorithm.

The exact times can be found in appendix B, in this section the results are visualized and presented to be easily interpreted. In some cases the results in a graph are normalized meaning that the running times are divided by the running times of the first algorithm in the graph. Specifically for the random games the running times vary a lot so normalizing is required to properly visualize the results.

15.1 Zielonka's family based

We compare the running times of the Zielonka's family based approaches with the Zielonka's product based approach. First we look at the model verification games.

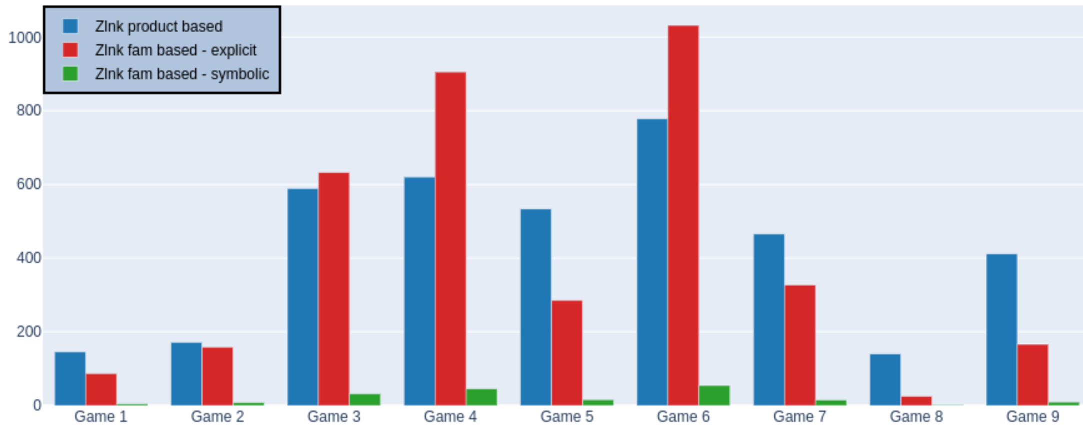


Figure 15: Running time of Zielonka's algorithms on the minepump problem

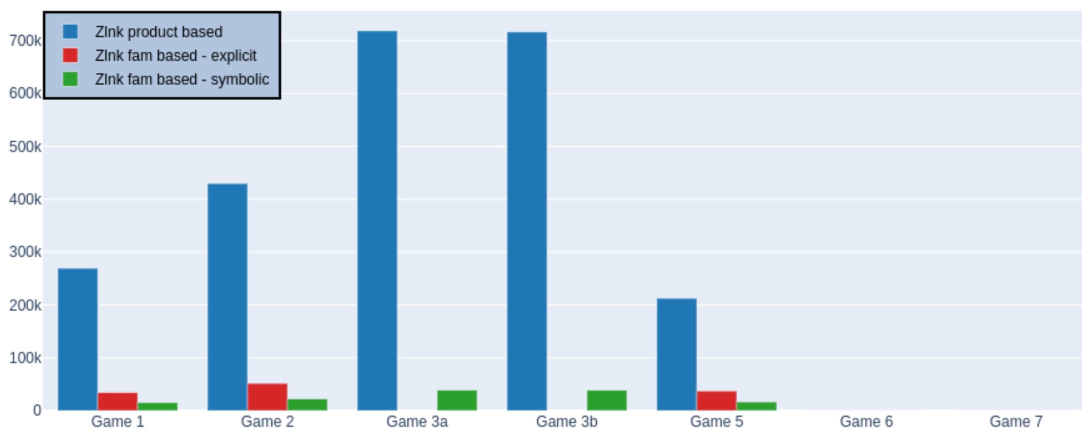


Figure 16: Running time of Zielonka's algorithms on the elevator problem

We can see that the performance of the explicit variant varies a lot between games. The symbolic variant greatly outperforms the product based approach for every problem.

Next we inspect the random games, first we look at the games with a variable λ and a random number of features. The graphs are normalized and the y-axis is cut off at 10.

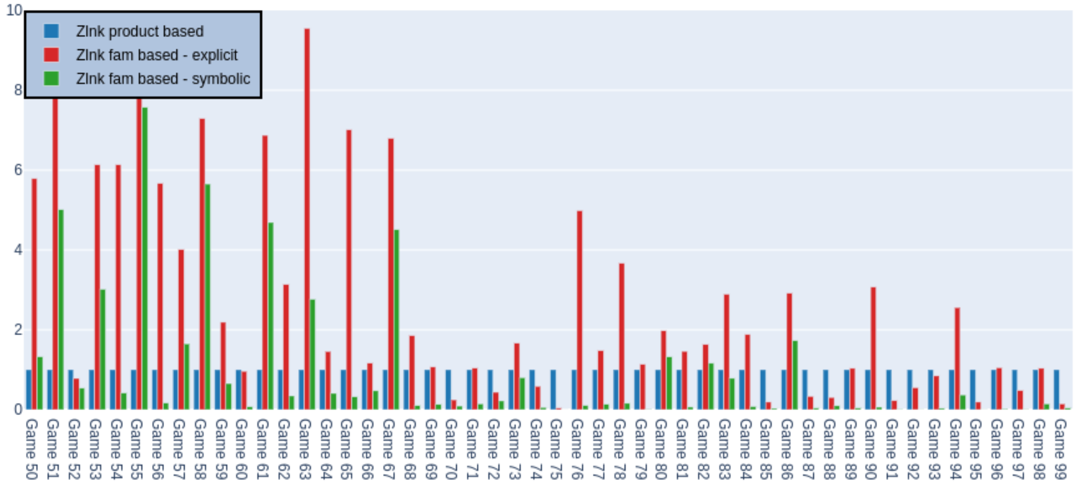


Figure 17: Running time of Zielonka's algorithms on randomgames of type 1 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

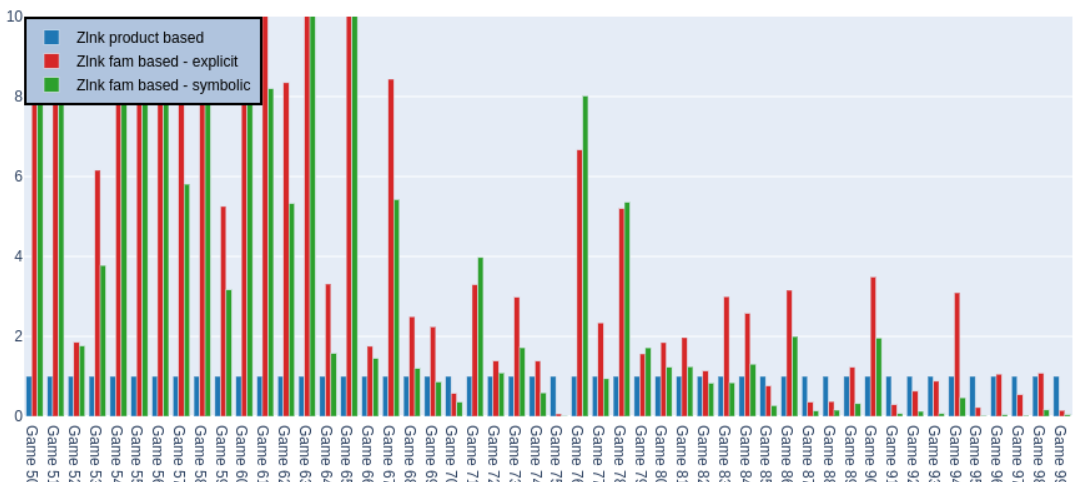


Figure 18: Running time of Zielonka's algorithms on randomgames of type 2 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

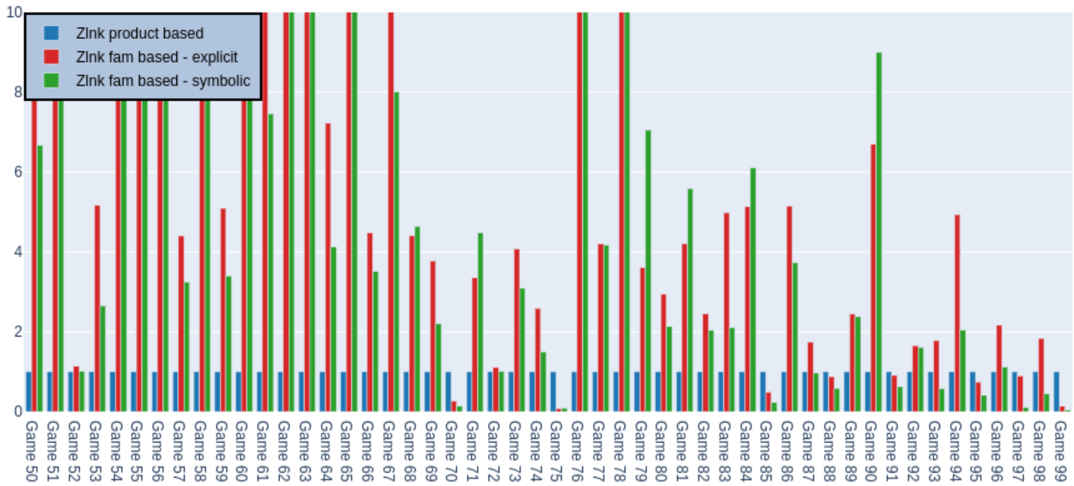


Figure 19: Running time of Zielonka's algorithms on randomgames of type 3 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

For type 1 games we see that when λ gets bigger the family based symbolic approach starts winning from the product based approach. There are a few exceptions to this, games 80, 82 and 86, all three of these games have only 4 features. As we will see later the less features there are in a game the worse the family based approaches perform.

For type 2 the explicit variant performs very similar to the type 1 games, however the symbolic approach performs much worse. This is due to the unstructured nature of the configuration sets which negatively influences bdd performance but has no effect on the explicit set operation. We also see the explicit algorithm outperforming the symbolic algorithm in some cases.

For type 3 games the product based approach performs generally better than the family based approaches unless λ becomes very high.

Next we inspect how the algorithms scale in terms of number of features

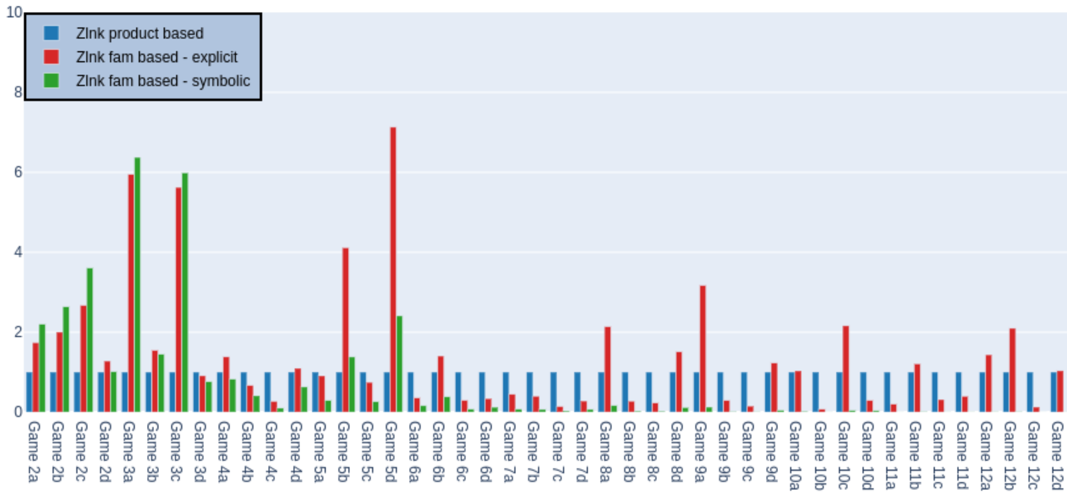


Figure 20: Running time of Zielonka's algorithms on randomgames of type 1 with $\lambda = 0.92$ and the number of features equal to the *game nr*, times are normalized and the y-axis is cut off at 10

We can clearly conclude that as the number of features increases the family based symbolic approach performs better compared to the product based approach.

Overall we can conclude that the explicit algorithm performs somewhat arbitrary, however the symbolic algorithm performs really well for model checking problems and random games that have similar properties. Also we can conclude that the algorithms scales well in terms of number of features.

15.2 Incremental pre-solve algorithm

We compare the running times of the incremental pre-solve approaches with the Zielonka's product based approach. First we look at the model verification games.

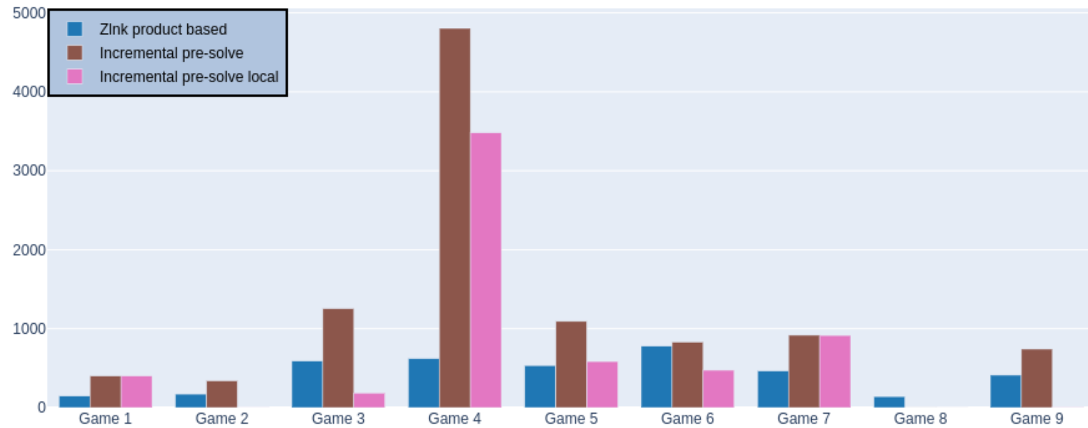


Figure 21: Running time of Zielonka's algorithms on the minepump problem

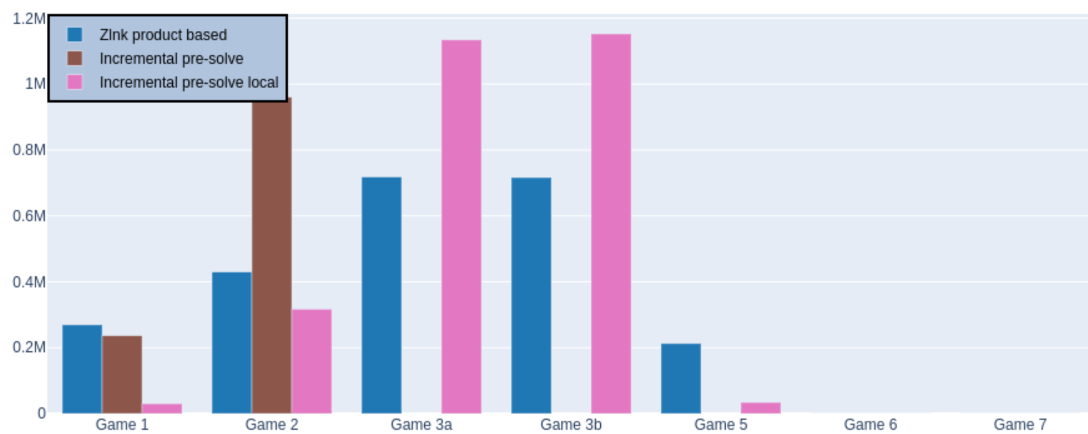


Figure 22: Running time of the incremental pre-solve algorithms on the elevator problem

The performance of the incremental pre-solve algorithm is generally worse that the product based approach, we do see however that the local variant is better in some cases.

Next we inspect the random games, first we look at the games with a variable λ and a random number of features. The graphs are normalized and the y-axis is cut off at 10.

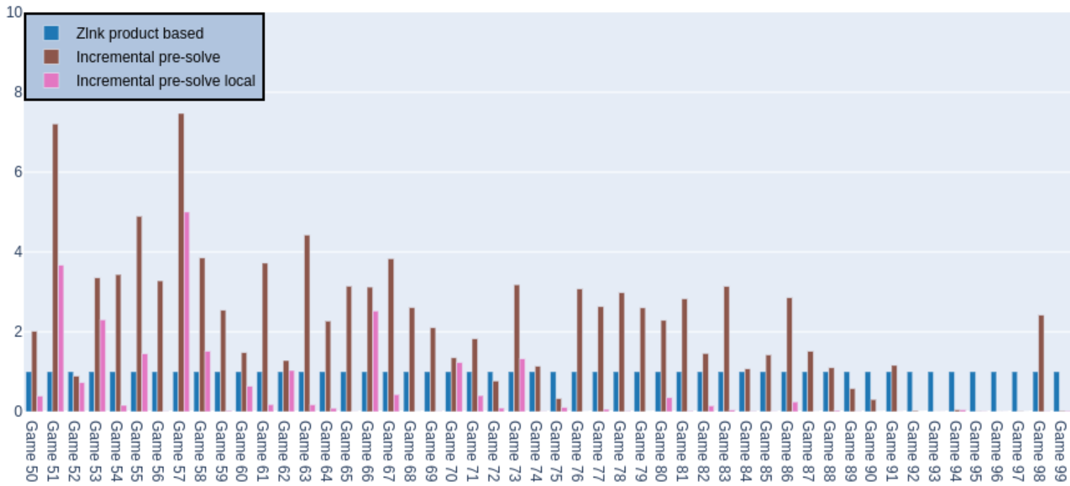


Figure 23: Running time of the incremental pre-solve algorithms on randomgames of type 1 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

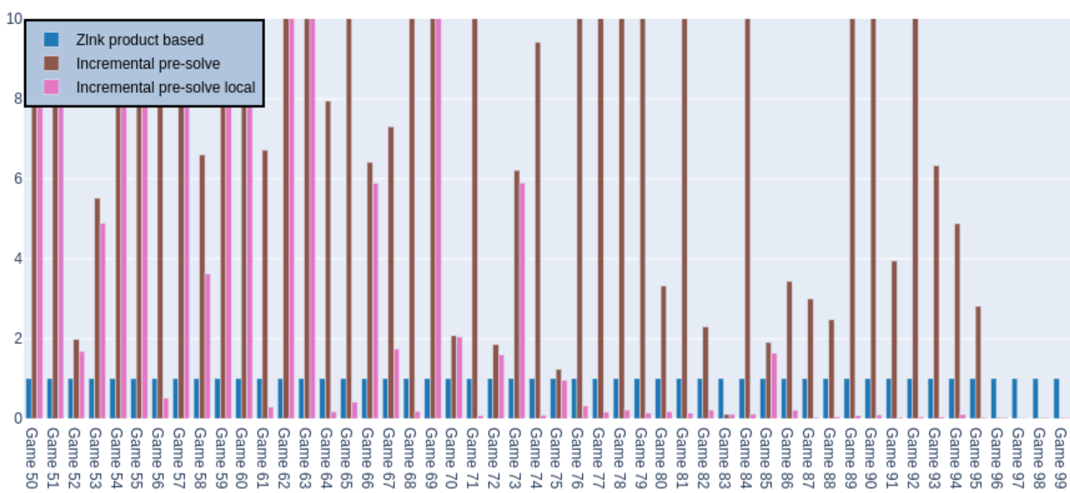


Figure 24: Running time of incremental pre-solve algorithms on randomgames of type 2 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

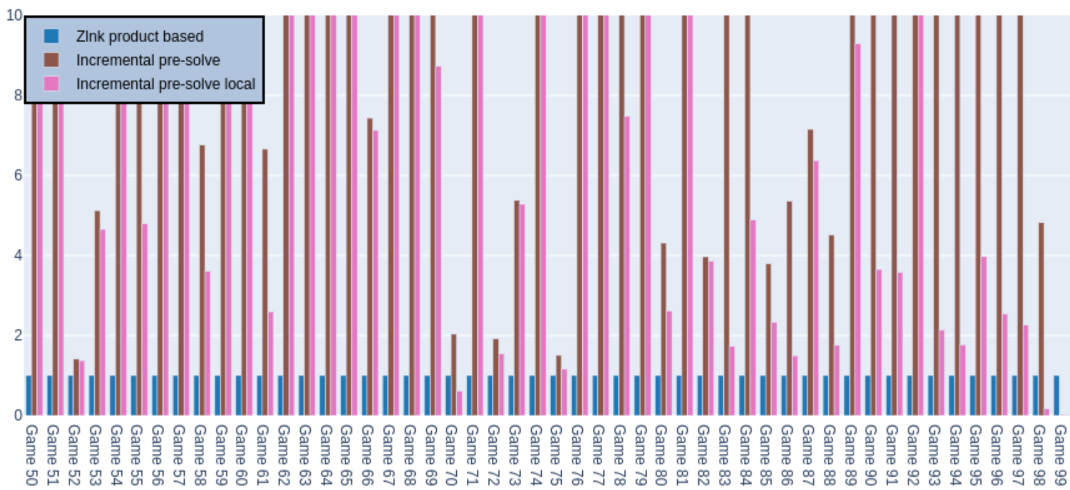


Figure 25: Running time of incremental pre-solve algorithms on randomgames of type 3 with $\lambda = \frac{game\ nr}{100}$, times are normalized and the y-axis is cut off at 10

For type 1 games we see that when λ gets bigger the family based approaches start performing better, specifically the local variant.

For type 2 the family based local approach still performs quite well when λ gets bigger. For type 3 games the product based approach outperforms the family based approaches.

Next we inspect how the algorithms scales in terms of number of features

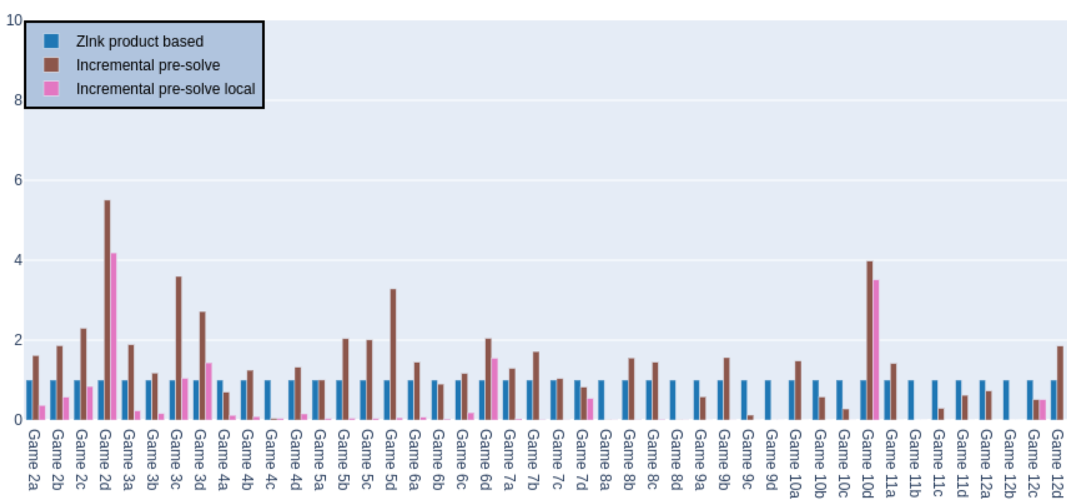


Figure 26: Running time of incremental pre-solve algorithms on randomgames of type 1 with $\lambda = 0.92$ and the number of features equal to the *game nr*, times are normalized and the y-axis is cut off at 10

The number of features seems to have little effect on the performance of the family based approaches.

Overall we can see that the local variant performs significantly better, specifically for random games. The local approach seems to be hit or miss where in some cases it finds vertex 0 quickly without traversing the tree but in other cases it has little effect compared to the global variant. Model verification problems generally have a vertex 0 that depends on a large part of the game and are therefore not very suitable to be solved locally. (TODO cite oid)

Appendices

A Auxiliary theorems and lemma's

Lemma A.1. *For $d, n \in \mathbb{N}$ with $d > 1$ and $n \geq 0$ the following inequality holds:*

$$(n + d - 1)^d + n + 1 \leq (n + d)^d$$

Proof. We expand the inequality.

$$(n + d - 1)^d + n + 1 \leq (n + d)^d$$
$$(n + d - 1)(n + d - 1)^{d-1} + n + 1 \leq (n + d)(n + d)^{d-1}$$
$$n(n + d - 1)^{d-1} + d(n + d - 1)^{d-1} - (n + d - 1)^{d-1} + n + 1 \leq n(n + d)^{d-1} + d(n + d)^{d-1}$$

Since $d > 1$ and $n \geq 0$ we can see that the left hand term $n(n + d - 1)^{d-1}$ is less or equal to the right hand term $n(n + d)^{d-1}$, similarly the left hand term $d(n + d - 1)^{d-1}$ is less or equal to the right hand term $d(n + d)^{d-1}$. Finally the term $(n + d - 1)^{d-1} \geq (n + 1)^{d-1} \geq n + 1$ and therefore $-(n + d - 1)^{d-1} + n + 1 \leq 0$. This proves the lemma. ☐

B Running time results

B.1 minepump

	Zlnk product based
Game 1	145.710686 ms
Game 2	171.767544 ms
Game 3	589.163537 ms
Game 4	620.295945 ms
Game 5	533.288498 ms
Game 6	777.771697 ms
Game 7	465.356122 ms
Game 8	139.804051 ms
Game 9	411.34498 ms

	Fixed-point product based
Game 1	130.587291 ms
Game 2	135.382598 ms
Game 3	460.184224 ms
Game 4	37898.67001 ms
Game 5	1346.308334 ms
Game 6	928.083238 ms
Game 7	669.829938 ms
Game 8	80.328725 ms
Game 9	823.229266 ms

	Fixed-point local product based
Game 1	66.477775 ms
Game 2	82.189287 ms
Game 3	278.245894 ms
Game 4	37764.888965 ms
Game 5	833.712816 ms
Game 6	544.744721 ms
Game 7	562.618467 ms
Game 8	8.503434 ms
Game 9	75.415805 ms

	Zlnk fam based - explicit
Game 1	87.111158 ms
Game 2	158.648592 ms
Game 3	632.834564 ms
Game 4	905.322212 ms
Game 5	285.163541 ms
Game 6	1031.566116 ms
Game 7	326.632454 ms
Game 8	25.146327 ms
Game 9	166.074354 ms

	Zlnk fam based - symbolic
Game 1	5.052944 ms
Game 2	8.372276 ms
Game 3	32.35237 ms
Game 4	45.629408 ms
Game 5	16.072678 ms
Game 6	54.307865 ms
Game 7	15.43434 ms
Game 8	2.104151 ms
Game 9	9.604102 ms

	Incremental pre-solve
Game 1	402.898316 ms
Game 2	341.198461 ms
Game 3	1255.949764 ms
Game 4	4803.836362 ms
Game 5	1092.277622 ms
Game 6	831.198424 ms
Game 7	917.083359 ms
Game 8	1.761178 ms
Game 9	742.183235 ms

	Incremental pre-solve local
Game 1	402.728056 ms
Game 2	1.827667 ms
Game 3	184.726048 ms
Game 4	3480.033761 ms
Game 5	583.474132 ms
Game 6	473.593019 ms
Game 7	914.134888 ms
Game 8	1.174847 ms
Game 9	4.019017 ms

B.2 elevator

	Zlnk product based
Game 1	269357.17363 ms
Game 2	429856.472992 ms
Game 3a	718657.056545 ms
Game 3b	716411.123774 ms
Game 5	212494.552309 ms
Game 6	0.260769 ms
Game 7	0.260798 ms

	Fixed-point product based
Game 1	880895.014563 ms
Game 2	8790000.10349 ms
Game 3a	29387446.6626 ms
Game 3b	29423372.371 ms
Game 5	418128.687347 ms
Game 6	0.182554 ms
Game 7	0.182125 ms

	Fixed-point local product based
Game 1	423068.957211 ms
Game 2	1983814.2006 ms
Game 3a	4635988.08945 ms
Game 3b	4666038.77282 ms
Game 5	282602.487182 ms
Game 6	0.11491 ms
Game 7	0.111502 ms

	Zlnk fam based - explicit
Game 1	33879.405405 ms
Game 2	51596.003539 ms
Game 3a	†
Game 3b	†
Game 5	37210.889792 ms
Game 6	0.099443 ms
Game 7	0.120632 ms

	Zlnk fam based - symbolic
Game 1	14964.371693 ms
Game 2	21803.007101 ms
Game 3a	38288.893506 ms
Game 3b	38213.782824 ms
Game 5	16324.397915 ms
Game 6	0.012454 ms
Game 7	0.012368 ms

	Incremental pre-solve
Game 1	236171.44533 ms
Game 2	960813.557869 ms
Game 3a	†
Game 3b	†
Game 5	†
Game 6	0.011157 ms
Game 7	0.0103 ms

	Incremental pre-solve local
Game 1	29941.840644 ms
Game 2	317217.608246 ms
Game 3a	1133904.4743 ms
Game 3b	1152374.57835 ms
Game 5	33845.973375 ms
Game 6	0.010736 ms
Game 7	0.010461 ms

B.3 FF randomgames

	Zlnk product based		Fixed-point product based		Fixed-point local product based
Game 50	22.894099 ms	Game 50	20.566329 ms	Game 50	20.595845 ms
Game 51	415.090564 ms	Game 51	6744.284861 ms	Game 51	6292.631423 ms
Game 52	3.531167 ms	Game 52	1.47327 ms	Game 52	1.398527 ms
Game 53	2.861018 ms	Game 53	4.295792 ms	Game 53	4.302088 ms
Game 54	94.868137 ms	Game 54	29.14741 ms	Game 54	29.24289 ms
Game 55	6.835354 ms	Game 55	14.425234 ms	Game 55	7.193027 ms
Game 56	469.55704 ms	Game 56	912.128577 ms	Game 56	472.727188 ms
Game 57	19.217615 ms	Game 57	199.442715 ms	Game 57	169.646163 ms
Game 58	5.671594 ms	Game 58	6.650932 ms	Game 58	6.664684 ms
Game 59	21.520462 ms	Game 59	30.331524 ms	Game 59	10.358405 ms
Game 60	82.895336 ms	Game 60	266.762206 ms	Game 60	267.307192 ms
Game 61	6.332443 ms	Game 61	2.756214 ms	Game 61	1.508336 ms
Game 62	85.507286 ms	Game 62	215.868882 ms	Game 62	205.976637 ms
Game 63	46.017808 ms	Game 63	226.614402 ms	Game 63	130.156094 ms
Game 64	15.773509 ms	Game 64	83.090986 ms	Game 64	29.198676 ms
Game 65	370.805409 ms	Game 65	266.050094 ms	Game 65	197.182221 ms
Game 66	22.776748 ms	Game 66	121.173001 ms	Game 66	120.811351 ms
Game 67	7.120147 ms	Game 67	30.477553 ms	Game 67	20.417376 ms
Game 68	81.248511 ms	Game 68	43.78565 ms	Game 68	18.646781 ms
Game 69	38.474281 ms	Game 69	28.032054 ms	Game 69	28.136298 ms
Game 70	2.863258 ms	Game 70	0.396067 ms	Game 70	0.381035 ms
Game 71	62.639661 ms	Game 71	28.951018 ms	Game 71	27.067918 ms
Game 72	0.843288 ms	Game 72	0.200488 ms	Game 72	0.204019 ms
Game 73	5.831069 ms	Game 73	10.418019 ms	Game 73	3.462291 ms
Game 74	48.460654 ms	Game 74	143.527764 ms	Game 74	143.903704 ms
Game 75	147.861501 ms	Game 75	46.377756 ms	Game 75	36.805055 ms
Game 76	461.039 ms	Game 76	173.248219 ms	Game 76	167.128798 ms
Game 77	96.339976 ms	Game 77	1023.572304 ms	Game 77	383.250688 ms
Game 78	383.656332 ms	Game 78	1500.871069 ms	Game 78	564.751313 ms
Game 79	755.153118 ms	Game 79	473.520308 ms	Game 79	473.792807 ms
Game 80	2.733223 ms	Game 80	1.31479 ms	Game 80	1.086947 ms
Game 81	185.213669 ms	Game 81	1054.141901 ms	Game 81	1053.657412 ms
Game 82	1.475768 ms	Game 82	1.317754 ms	Game 82	1.324338 ms
Game 83	22.639925 ms	Game 83	24.022014 ms	Game 83	14.837088 ms
Game 84	193.419532 ms	Game 84	947.266091 ms	Game 84	357.204857 ms
Game 85	14.283935 ms	Game 85	9.411088 ms	Game 85	9.476771 ms
Game 86	6.635476 ms	Game 86	17.27655 ms	Game 86	7.262061 ms
Game 87	17.298111 ms	Game 87	5.299216 ms	Game 87	5.352538 ms
Game 88	7.915332 ms	Game 88	5.561753 ms	Game 88	2.610671 ms
Game 89	246.627127 ms	Game 89	1103.733905 ms	Game 89	543.43552 ms
Game 90	631.734621 ms	Game 90	642.737385 ms	Game 90	367.617943 ms
Game 91	123.777906 ms	Game 91	210.113964 ms	Game 91	87.154413 ms
Game 92	465.960801 ms	Game 92	140.735555 ms	Game 92	142.924215 ms
Game 93	117.739068 ms	Game 93	124.311004 ms	Game 93	74.032315 ms
Game 94	98.094232 ms	Game 94	465.467471 ms	Game 94	465.642971 ms
Game 95	227.341992 ms	Game 95	1940.174842 ms	Game 95	1939.324955 ms
Game 96	379.466667 ms	Game 96	1066.142203 ms	Game 96	1068.923695 ms
Game 97	131.677711 ms	Game 97	202.031576 ms	Game 97	202.097029 ms
Game 98	35.736698 ms	Game 98	24.381076 ms	Game 98	24.263109 ms
Game 99	1.010203 ms	Game 99	0.153665 ms	Game 99	0.154271 ms

	Zlnk fam based - explicit
Game 50	132.471665 ms
Game 51	9419.058069 ms
Game 52	2.7618 ms
Game 53	17.544563 ms
Game 54	581.81683 ms
Game 55	71.685999 ms
Game 56	2661.310318 ms
Game 57	77.107864 ms
Game 58	41.351346 ms
Game 59	47.101189 ms
Game 60	79.691535 ms
Game 61	43.512465 ms
Game 62	268.348187 ms
Game 63	439.554763 ms
Game 64	23.015237 ms
Game 65	2598.745644 ms
Game 66	26.532439 ms
Game 67	48.409873 ms
Game 68	150.920555 ms
Game 69	41.092679 ms
Game 70	0.704051 ms
Game 71	65.254112 ms
Game 72	0.367596 ms
Game 73	9.710427 ms
Game 74	28.357716 ms
Game 75	6.084672 ms
Game 76	2299.390861 ms
Game 77	142.898393 ms
Game 78	1406.919686 ms
Game 79	858.03388 ms
Game 80	5.400674 ms
Game 81	270.81002 ms
Game 82	2.407317 ms
Game 83	65.527967 ms
Game 84	363.863434 ms
Game 85	2.711993 ms
Game 86	19.38862 ms
Game 87	5.777005 ms
Game 88	2.419001 ms
Game 89	255.358705 ms
Game 90	1938.833122 ms
Game 91	27.914326 ms
Game 92	257.34058 ms
Game 93	99.616438 ms
Game 94	250.807913 ms
Game 95	43.182079 ms
Game 96	399.435036 ms
Game 97	63.44707 ms
Game 98	36.997553 ms
Game 99	0.147938 ms

	Zlnk fam based - symbolic
Game 50	30.411701 ms
Game 51	2079.058824 ms
Game 52	1.899092 ms
Game 53	8.634299 ms
Game 54	39.92786 ms
Game 55	51.745846 ms
Game 56	78.389071 ms
Game 57	31.689397 ms
Game 58	32.055879 ms
Game 59	14.083391 ms
Game 60	5.963575 ms
Game 61	29.680534 ms
Game 62	29.427663 ms
Game 63	127.371121 ms
Game 64	6.433852 ms
Game 65	118.75719 ms
Game 66	10.763784 ms
Game 67	32.088827 ms
Game 68	8.20797 ms
Game 69	5.009418 ms
Game 70	0.259005 ms
Game 71	9.179291 ms
Game 72	0.184988 ms
Game 73	4.683395 ms
Game 74	2.693721 ms
Game 75	0.266035 ms
Game 76	50.078993 ms
Game 77	12.918694 ms
Game 78	61.057056 ms
Game 79	9.95944 ms
Game 80	3.61149 ms
Game 81	12.008958 ms
Game 82	1.714712 ms
Game 83	17.773642 ms
Game 84	15.673699 ms
Game 85	0.463345 ms
Game 86	11.490767 ms
Game 87	0.748539 ms
Game 88	0.770723 ms
Game 89	10.615619 ms
Game 90	39.17593 ms
Game 91	1.540775 ms
Game 92	2.786512 ms
Game 93	3.99316 ms
Game 94	36.186725 ms
Game 95	1.118923 ms
Game 96	7.881691 ms
Game 97	1.340903 ms
Game 98	5.233279 ms
Game 99	0.047058 ms

	Incremental pre-solve
Game 50	46.080898 ms
Game 51	2992.042268 ms
Game 52	3.151228 ms
Game 53	9.583558 ms
Game 54	325.711561 ms
Game 55	33.458447 ms
Game 56	1539.005235 ms
Game 57	143.532033 ms
Game 58	21.824657 ms
Game 59	54.624111 ms
Game 60	122.370786 ms
Game 61	23.554931 ms
Game 62	109.501093 ms
Game 63	203.673008 ms
Game 64	35.808988 ms
Game 65	1165.840461 ms
Game 66	71.074647 ms
Game 67	27.291058 ms
Game 68	212.068661 ms
Game 69	80.825415 ms
Game 70	3.859968 ms
Game 71	114.442028 ms
Game 72	0.649724 ms
Game 73	18.543163 ms
Game 74	55.220584 ms
Game 75	48.855116 ms
Game 76	1419.192729 ms
Game 77	253.275833 ms
Game 78	1142.785118 ms
Game 79	1962.348341 ms
Game 80	6.247378 ms
Game 81	524.071394 ms
Game 82	2.152315 ms
Game 83	71.00827 ms
Game 84	206.907479 ms
Game 85	20.351748 ms
Game 86	18.925971 ms
Game 87	26.118424 ms
Game 88	8.694553 ms
Game 89	143.203366 ms
Game 90	192.609233 ms
Game 91	143.36905 ms
Game 92	11.604601 ms
Game 93	0.505494 ms
Game 94	4.989098 ms
Game 95	2.18487 ms
Game 96	1.447915 ms
Game 97	1.369827 ms
Game 98	86.552119 ms
Game 99	0.02177 ms

	Incremental pre-solve local
Game 50	9.018887 ms
Game 51	1523.362457 ms
Game 52	2.56276 ms
Game 53	6.587165 ms
Game 54	14.87686 ms
Game 55	9.903866 ms
Game 56	4.660675 ms
Game 57	96.08021 ms
Game 58	8.569419 ms
Game 59	0.534212 ms
Game 60	52.722595 ms
Game 61	1.132779 ms
Game 62	88.117407 ms
Game 63	8.078743 ms
Game 64	1.370403 ms
Game 65	2.78775 ms
Game 66	57.445895 ms
Game 67	3.053301 ms
Game 68	0.384612 ms
Game 69	0.376439 ms
Game 70	3.533979 ms
Game 71	25.211715 ms
Game 72	0.076533 ms
Game 73	7.696058 ms
Game 74	0.464487 ms
Game 75	15.709783 ms
Game 76	1.285309 ms
Game 77	5.801185 ms
Game 78	6.765665 ms
Game 79	0.722754 ms
Game 80	0.953798 ms
Game 81	3.425409 ms
Game 82	0.217681 ms
Game 83	1.021352 ms
Game 84	2.749428 ms
Game 85	0.121502 ms
Game 86	1.616307 ms
Game 87	0.078561 ms
Game 88	0.224175 ms
Game 89	2.50544 ms
Game 90	1.621967 ms
Game 91	0.683825 ms
Game 92	0.294685 ms
Game 93	0.503414 ms
Game 94	4.977672 ms
Game 95	2.243018 ms
Game 96	1.475905 ms
Game 97	0.312385 ms
Game 98	0.45833 ms
Game 99	0.020862 ms

B.4 FC randomgames

	Zlnk product based
Game 50	21.231204 ms
Game 51	400.670829 ms
Game 52	2.543695 ms
Game 53	2.91148 ms
Game 54	92.385426 ms
Game 55	6.931507 ms
Game 56	457.498781 ms
Game 57	18.410363 ms
Game 58	5.796614 ms
Game 59	20.884377 ms
Game 60	82.441126 ms
Game 61	5.77509 ms
Game 62	72.394361 ms
Game 63	47.917602 ms
Game 64	15.783667 ms
Game 65	364.392252 ms
Game 66	20.900057 ms
Game 67	6.489432 ms
Game 68	81.398073 ms
Game 69	38.602005 ms
Game 70	2.862779 ms
Game 71	61.316237 ms
Game 72	1.02396 ms
Game 73	4.991399 ms
Game 74	48.053337 ms
Game 75	84.6986 ms
Game 76	437.874416 ms
Game 77	97.581837 ms
Game 78	399.629215 ms
Game 79	747.373823 ms
Game 80	2.74305 ms
Game 81	194.11536 ms
Game 82	1.547212 ms
Game 83	23.34109 ms
Game 84	214.313591 ms
Game 85	18.135857 ms
Game 86	7.022848 ms
Game 87	17.597631 ms
Game 88	8.248575 ms
Game 89	248.802081 ms
Game 90	661.651541 ms
Game 91	126.08789 ms
Game 92	484.469978 ms
Game 93	124.135603 ms
Game 94	103.01206 ms
Game 95	233.848524 ms
Game 96	374.349302 ms
Game 97	141.311059 ms
Game 98	35.467827 ms
Game 99	1.021463 ms

	Fixed-point product based
Game 50	25.788145 ms
Game 51	6841.202002 ms
Game 52	1.142889 ms
Game 53	3.264741 ms
Game 54	30.670339 ms
Game 55	10.21778 ms
Game 56	1312.727673 ms
Game 57	336.312221 ms
Game 58	3.993694 ms
Game 59	19.763114 ms
Game 60	419.528973 ms
Game 61	1.847166 ms
Game 62	157.99114 ms
Game 63	123.032105 ms
Game 64	50.739421 ms
Game 65	339.604908 ms
Game 66	55.454246 ms
Game 67	56.577501 ms
Game 68	35.266073 ms
Game 69	31.136878 ms
Game 70	0.623003 ms
Game 71	28.043222 ms
Game 72	0.720565 ms
Game 73	14.39249 ms
Game 74	233.078601 ms
Game 75	20.684427 ms
Game 76	156.408315 ms
Game 77	411.702241 ms
Game 78	798.928246 ms
Game 79	463.166073 ms
Game 80	1.714026 ms
Game 81	735.675399 ms
Game 82	1.853504 ms
Game 83	14.961404 ms
Game 84	462.353075 ms
Game 85	18.109145 ms
Game 86	10.22887 ms
Game 87	4.766618 ms
Game 88	4.285175 ms
Game 89	569.014841 ms
Game 90	1137.612069 ms
Game 91	380.726518 ms
Game 92	159.622577 ms
Game 93	190.273961 ms
Game 94	595.608117 ms
Game 95	3054.323181 ms
Game 96	1172.194676 ms
Game 97	120.681324 ms
Game 98	27.819839 ms
Game 99	0.184036 ms

	Fixed-point local product based
Game 50	23.491823 ms
Game 51	6358.738676 ms
Game 52	1.120596 ms
Game 53	1.841316 ms
Game 54	16.842579 ms
Game 55	5.598839 ms
Game 56	1316.381878 ms
Game 57	323.38378 ms
Game 58	2.154177 ms
Game 59	8.593163 ms
Game 60	266.971185 ms
Game 61	0.974743 ms
Game 62	153.270569 ms
Game 63	123.573552 ms
Game 64	50.76468 ms
Game 65	151.33958 ms
Game 66	43.175666 ms
Game 67	56.524643 ms
Game 68	35.68747 ms
Game 69	27.232735 ms
Game 70	0.42265 ms
Game 71	28.190491 ms
Game 72	0.384138 ms
Game 73	12.369724 ms
Game 74	86.760695 ms
Game 75	20.558411 ms
Game 76	71.018835 ms
Game 77	412.161454 ms
Game 78	800.264776 ms
Game 79	462.523326 ms
Game 80	0.923882 ms
Game 81	735.152819 ms
Game 82	1.870616 ms
Game 83	9.113067 ms
Game 84	295.219217 ms
Game 85	11.739621 ms
Game 86	10.344082 ms
Game 87	4.326818 ms
Game 88	2.003342 ms
Game 89	569.378285 ms
Game 90	1133.058396 ms
Game 91	225.493779 ms
Game 92	160.648632 ms
Game 93	107.219647 ms
Game 94	596.556856 ms
Game 95	3053.310575 ms
Game 96	1175.237651 ms
Game 97	76.270106 ms
Game 98	27.717006 ms
Game 99	0.168606 ms

	Zlnk fam based - explicit
Game 50	413.757965 ms
Game 51	27378.580718 ms
Game 52	4.704715 ms
Game 53	17.924588 ms
Game 54	1898.715622 ms
Game 55	116.488589 ms
Game 56	6597.670929 ms
Game 57	151.524968 ms
Game 58	70.223866 ms
Game 59	109.666063 ms
Game 60	1268.010613 ms
Game 61	70.471579 ms
Game 62	604.247392 ms
Game 63	798.972653 ms
Game 64	52.263141 ms
Game 65	5397.140781 ms
Game 66	36.613242 ms
Game 67	54.748121 ms
Game 68	202.996454 ms
Game 69	86.514338 ms
Game 70	1.632376 ms
Game 71	202.272811 ms
Game 72	1.422324 ms
Game 73	14.89502 ms
Game 74	66.366483 ms
Game 75	5.1388 ms
Game 76	2919.703498 ms
Game 77	228.166063 ms
Game 78	2078.84805 ms
Game 79	1167.145606 ms
Game 80	5.059871 ms
Game 81	382.166334 ms
Game 82	1.760136 ms
Game 83	69.88873 ms
Game 84	552.858032 ms
Game 85	13.873269 ms
Game 86	22.161829 ms
Game 87	6.245994 ms
Game 88	3.055452 ms
Game 89	304.971561 ms
Game 90	2308.603817 ms
Game 91	36.671807 ms
Game 92	304.780275 ms
Game 93	108.993149 ms
Game 94	318.492416 ms
Game 95	51.491158 ms
Game 96	395.008307 ms
Game 97	76.947702 ms
Game 98	38.345677 ms
Game 99	0.151832 ms

	Zlnk fam based - symbolic
Game 50	200.692363 ms
Game 51	93954.150841 ms
Game 52	4.484061 ms
Game 53	10.992585 ms
Game 54	3785.109453 ms
Game 55	81.195372 ms
Game 56	15885.110636 ms
Game 57	106.939326 ms
Game 58	46.996476 ms
Game 59	66.098965 ms
Game 60	1037.37052 ms
Game 61	47.343002 ms
Game 62	385.736614 ms
Game 63	600.720464 ms
Game 64	24.852818 ms
Game 65	9002.50055 ms
Game 66	30.259984 ms
Game 67	35.210879 ms
Game 68	97.369306 ms
Game 69	33.332137 ms
Game 70	1.019247 ms
Game 71	243.616234 ms
Game 72	1.111766 ms
Game 73	8.567746 ms
Game 74	28.256644 ms
Game 75	1.396665 ms
Game 76	3511.128935 ms
Game 77	91.683859 ms
Game 78	2141.834396 ms
Game 79	1281.847449 ms
Game 80	3.367123 ms
Game 81	241.019711 ms
Game 82	1.286789 ms
Game 83	19.583566 ms
Game 84	279.733505 ms
Game 85	4.912336 ms
Game 86	14.018757 ms
Game 87	2.353829 ms
Game 88	1.272726 ms
Game 89	78.647224 ms
Game 90	1296.34402 ms
Game 91	8.449923 ms
Game 92	60.059893 ms
Game 93	8.330845 ms
Game 94	47.843447 ms
Game 95	4.871075 ms
Game 96	16.973221 ms
Game 97	3.288376 ms
Game 98	5.675021 ms
Game 99	0.045707 ms

	Incremental pre-solve
Game 50	274.513552 ms
Game 51	61404.869851 ms
Game 52	5.041807 ms
Game 53	16.068485 ms
Game 54	8252.836326 ms
Game 55	69.251691 ms
Game 56	88278.874709 ms
Game 57	666.755859 ms
Game 58	38.229257 ms
Game 59	226.141247 ms
Game 60	2514.769282 ms
Game 61	38.780713 ms
Game 62	1176.272233 ms
Game 63	858.516357 ms
Game 64	125.34107 ms
Game 65	37422.726912 ms
Game 66	133.928952 ms
Game 67	47.395902 ms
Game 68	2787.521288 ms
Game 69	512.772417 ms
Game 70	5.953089 ms
Game 71	757.297336 ms
Game 72	1.893045 ms
Game 73	30.999393 ms
Game 74	452.229702 ms
Game 75	104.657565 ms
Game 76	55820.453259 ms
Game 77	1661.170935 ms
Game 78	20559.171143 ms
Game 79	74687.630715 ms
Game 80	9.092573 ms
Game 81	5052.554785 ms
Game 82	3.555746 ms
Game 83	2.529063 ms
Game 84	5286.51502 ms
Game 85	34.527788 ms
Game 86	24.111287 ms
Game 87	52.726666 ms
Game 88	20.382232 ms
Game 89	3639.537318 ms
Game 90	24874.107362 ms
Game 91	496.947157 ms
Game 92	13755.969387 ms
Game 93	785.924095 ms
Game 94	502.19285 ms
Game 95	656.998381 ms
Game 96	8.802605 ms
Game 97	1.757358 ms
Game 98	0.72427 ms
Game 99	0.020685 ms

	Incremental pre-solve local
Game 50	269.718821 ms
Game 51	60355.20469 ms
Game 52	4.278918 ms
Game 53	14.222552 ms
Game 54	7967.297797 ms
Game 55	57.062669 ms
Game 56	231.673885 ms
Game 57	638.551196 ms
Game 58	20.990244 ms
Game 59	185.289698 ms
Game 60	2338.294662 ms
Game 61	1.645522 ms
Game 62	1120.42437 ms
Game 63	697.99699 ms
Game 64	2.75111 ms
Game 65	149.314436 ms
Game 66	122.997515 ms
Game 67	11.312594 ms
Game 68	14.694391 ms
Game 69	461.680203 ms
Game 70	5.849632 ms
Game 71	4.622708 ms
Game 72	1.631972 ms
Game 73	29.434576 ms
Game 74	4.10057 ms
Game 75	81.1602 ms
Game 76	137.661231 ms
Game 77	15.629775 ms
Game 78	85.947227 ms
Game 79	101.015369 ms
Game 80	0.48114 ms
Game 81	25.734733 ms
Game 82	0.330751 ms
Game 83	2.53277 ms
Game 84	24.373237 ms
Game 85	29.590797 ms
Game 86	1.480927 ms
Game 87	0.514867 ms
Game 88	0.38692 ms
Game 89	18.561496 ms
Game 90	60.996587 ms
Game 91	3.65302 ms
Game 92	20.437045 ms
Game 93	4.298427 ms
Game 94	10.100405 ms
Game 95	6.47525 ms
Game 96	8.752235 ms
Game 97	1.736223 ms
Game 98	0.749523 ms
Game 99	0.020884 ms

B.5 BC randomgames

	Zlnk product based		Fixed-point product based		Fixed-point local product based
Game 50	20.298282 ms	Game 50	33.39416 ms	Game 50	13.808627 ms
Game 51	389.234391 ms	Game 51	5534.871778 ms	Game 51	5326.973496 ms
Game 52	3.381146 ms	Game 52	1.26391 ms	Game 52	1.276611 ms
Game 53	2.74789 ms	Game 53	3.170443 ms	Game 53	1.64824 ms
Game 54	91.837746 ms	Game 54	26.331521 ms	Game 54	12.136227 ms
Game 55	6.545738 ms	Game 55	11.795348 ms	Game 55	6.576087 ms
Game 56	458.625377 ms	Game 56	1504.589399 ms	Game 56	944.411779 ms
Game 57	19.114806 ms	Game 57	191.552096 ms	Game 57	191.884138 ms
Game 58	5.538301 ms	Game 58	5.875631 ms	Game 58	5.840379 ms
Game 59	21.573339 ms	Game 59	22.604068 ms	Game 59	10.144003 ms
Game 60	87.188907 ms	Game 60	420.111746 ms	Game 60	410.714964 ms
Game 61	5.934875 ms	Game 61	2.346122 ms	Game 61	1.14639 ms
Game 62	68.291827 ms	Game 62	210.076795 ms	Game 62	205.434014 ms
Game 63	42.398987 ms	Game 63	268.083218 ms	Game 63	267.867202 ms
Game 64	15.568307 ms	Game 64	188.800481 ms	Game 64	77.285434 ms
Game 65	347.61841 ms	Game 65	246.453054 ms	Game 65	223.400633 ms
Game 66	28.035516 ms	Game 66	109.679181 ms	Game 66	106.379365 ms
Game 67	6.988114 ms	Game 67	24.159304 ms	Game 67	13.235959 ms
Game 68	86.013404 ms	Game 68	32.304285 ms	Game 68	32.293533 ms
Game 69	39.55494 ms	Game 69	35.825679 ms	Game 69	35.856809 ms
Game 70	2.879269 ms	Game 70	0.344868 ms	Game 70	0.342402 ms
Game 71	62.718275 ms	Game 71	21.781779 ms	Game 71	20.189885 ms
Game 72	0.947918 ms	Game 72	0.602684 ms	Game 72	0.587294 ms
Game 73	4.984298 ms	Game 73	7.749658 ms	Game 73	6.687538 ms
Game 74	49.458599 ms	Game 74	212.354638 ms	Game 74	209.573048 ms
Game 75	111.674596 ms	Game 75	25.740094 ms	Game 75	24.833401 ms
Game 76	462.571733 ms	Game 76	154.448659 ms	Game 76	154.408502 ms
Game 77	99.018306 ms	Game 77	756.884077 ms	Game 77	301.337573 ms
Game 78	370.784036 ms	Game 78	1174.479264 ms	Game 78	1173.799697 ms
Game 79	775.469857 ms	Game 79	422.807031 ms	Game 79	220.204762 ms
Game 80	2.814515 ms	Game 80	1.094506 ms	Game 80	1.11143 ms
Game 81	190.327957 ms	Game 81	934.68797 ms	Game 81	414.742317 ms
Game 82	1.573491 ms	Game 82	2.723153 ms	Game 82	2.324309 ms
Game 83	22.376837 ms	Game 83	18.059568 ms	Game 83	10.075222 ms
Game 84	199.532348 ms	Game 84	414.709718 ms	Game 84	415.978428 ms
Game 85	15.270509 ms	Game 85	20.150961 ms	Game 85	15.497647 ms
Game 86	6.841671 ms	Game 86	20.67089 ms	Game 86	8.184581 ms
Game 87	18.137133 ms	Game 87	5.526844 ms	Game 87	4.216409 ms
Game 88	7.893327 ms	Game 88	4.486084 ms	Game 88	4.532044 ms
Game 89	250.111449 ms	Game 89	792.944337 ms	Game 89	793.009891 ms
Game 90	651.623509 ms	Game 90	755.171693 ms	Game 90	448.992638 ms
Game 91	125.053158 ms	Game 91	380.795967 ms	Game 91	381.016753 ms
Game 92	490.923857 ms	Game 92	162.856618 ms	Game 92	77.706188 ms
Game 93	125.648658 ms	Game 93	131.000885 ms	Game 93	131.040994 ms
Game 94	108.680939 ms	Game 94	402.117363 ms	Game 94	402.395126 ms
Game 95	253.314478 ms	Game 95	3293.720234 ms	Game 95	3293.137131 ms
Game 96	391.934301 ms	Game 96	1700.935701 ms	Game 96	1060.421164 ms
Game 97	133.309666 ms	Game 97	80.308982 ms	Game 97	80.576536 ms
Game 98	36.558556 ms	Game 98	40.934591 ms	Game 98	23.666594 ms
Game 99	1.035994 ms	Game 99	0.13178 ms	Game 99	0.130765 ms

	Zlnk fam based - explicit
Game 50	272.87934 ms
Game 51	13770.686241 ms
Game 52	3.836656 ms
Game 53	14.209937 ms
Game 54	1010.766938 ms
Game 55	108.358441 ms
Game 56	6623.716743 ms
Game 57	84.169799 ms
Game 58	63.226013 ms
Game 59	109.845438 ms
Game 60	1598.560793 ms
Game 61	63.833808 ms
Game 62	887.779447 ms
Game 63	789.117049 ms
Game 64	112.455154 ms
Game 65	6539.720485 ms
Game 66	125.454045 ms
Game 67	76.752727 ms
Game 68	378.887843 ms
Game 69	149.327624 ms
Game 70	0.767725 ms
Game 71	210.212655 ms
Game 72	1.053727 ms
Game 73	20.318549 ms
Game 74	127.935471 ms
Game 75	8.465813 ms
Game 76	5821.111494 ms
Game 77	416.38086 ms
Game 78	3873.990312 ms
Game 79	2799.949264 ms
Game 80	8.277526 ms
Game 81	801.164588 ms
Game 82	3.862032 ms
Game 83	111.584134 ms
Game 84	1024.056176 ms
Game 85	7.472206 ms
Game 86	35.232936 ms
Game 87	31.6398 ms
Game 88	6.901048 ms
Game 89	610.708056 ms
Game 90	4366.502404 ms
Game 91	113.407261 ms
Game 92	811.153177 ms
Game 93	223.680626 ms
Game 94	536.192172 ms
Game 95	187.47273 ms
Game 96	847.743778 ms
Game 97	119.463323 ms
Game 98	67.089435 ms
Game 99	0.13689 ms

	Zlnk fam based - symbolic
Game 50	135.362033 ms
Game 51	37114.927305 ms
Game 52	3.409047 ms
Game 53	7.283814 ms
Game 54	1670.262453 ms
Game 55	78.635768 ms
Game 56	17153.985334 ms
Game 57	62.038888 ms
Game 58	45.695908 ms
Game 59	73.307484 ms
Game 60	1671.550569 ms
Game 61	44.284777 ms
Game 62	810.413904 ms
Game 63	673.405897 ms
Game 64	64.221201 ms
Game 65	13227.069691 ms
Game 66	98.544214 ms
Game 67	55.949771 ms
Game 68	398.622708 ms
Game 69	87.268526 ms
Game 70	0.408592 ms
Game 71	281.020599 ms
Game 72	0.951768 ms
Game 73	15.412638 ms
Game 74	73.971327 ms
Game 75	9.798386 ms
Game 76	10732.162612 ms
Game 77	413.371941 ms
Game 78	6407.769293 ms
Game 79	5470.55466 ms
Game 80	5.999003 ms
Game 81	1063.593036 ms
Game 82	3.203672 ms
Game 83	47.147434 ms
Game 84	1218.575473 ms
Game 85	3.488327 ms
Game 86	25.523141 ms
Game 87	17.55912 ms
Game 88	4.609721 ms
Game 89	594.837855 ms
Game 90	5863.378422 ms
Game 91	78.077081 ms
Game 92	790.793478 ms
Game 93	72.80851 ms
Game 94	222.219913 ms
Game 95	103.348175 ms
Game 96	438.409169 ms
Game 97	14.583179 ms
Game 98	16.186327 ms
Game 99	0.043466 ms

	Incremental pre-solve
Game 50	223.466021 ms
Game 51	47937.178942 ms
Game 52	4.769585 ms
Game 53	14.069063 ms
Game 54	6622.38747 ms
Game 55	60.795483 ms
Game 56	75834.067772 ms
Game 57	497.534014 ms
Game 58	37.426839 ms
Game 59	225.083612 ms
Game 60	2836.00766 ms
Game 61	39.508389 ms
Game 62	1499.171081 ms
Game 63	1077.431545 ms
Game 64	256.577525 ms
Game 65	42208.617471 ms
Game 66	208.42148 ms
Game 67	164.479026 ms
Game 68	4443.06214 ms
Game 69	742.156512 ms
Game 70	5.855942 ms
Game 71	1155.989937 ms
Game 72	1.819665 ms
Game 73	26.794277 ms
Game 74	996.17373 ms
Game 75	167.774753 ms
Game 76	99258.828158 ms
Game 77	3530.63729 ms
Game 78	34238.537422 ms
Game 79	143423.447357 ms
Game 80	12.132405 ms
Game 81	10427.95421 ms
Game 82	6.241549 ms
Game 83	255.292567 ms
Game 84	11249.112673 ms
Game 85	57.947425 ms
Game 86	36.634892 ms
Game 87	129.742853 ms
Game 88	35.592823 ms
Game 89	9632.501725 ms
Game 90	67689.612809 ms
Game 91	1946.877179 ms
Game 92	43094.739497 ms
Game 93	2865.112811 ms
Game 94	1356.038341 ms
Game 95	4543.837972 ms
Game 96	13465.080253 ms
Game 97	1906.935486 ms
Game 98	176.16669 ms
Game 99	0.020168 ms

	Incremental pre-solve local
Game 50	182.975723 ms
Game 51	44671.815935 ms
Game 52	4.637662 ms
Game 53	12.782434 ms
Game 54	4032.451405 ms
Game 55	31.408177 ms
Game 56	56605.457611 ms
Game 57	404.737995 ms
Game 58	19.957296 ms
Game 59	180.302945 ms
Game 60	2498.227687 ms
Game 61	15.375766 ms
Game 62	1364.469331 ms
Game 63	664.211163 ms
Game 64	205.724963 ms
Game 65	34218.480539 ms
Game 66	199.78587 ms
Game 67	136.205044 ms
Game 68	1704.763075 ms
Game 69	345.406447 ms
Game 70	1.766606 ms
Game 71	1111.678948 ms
Game 72	1.462348 ms
Game 73	26.320828 ms
Game 74	826.088687 ms
Game 75	129.308793 ms
Game 76	10589.753997 ms
Game 77	1986.122688 ms
Game 78	2772.260825 ms
Game 79	34775.330723 ms
Game 80	7.359347 ms
Game 81	3749.058872 ms
Game 82	6.064545 ms
Game 83	38.694408 ms
Game 84	974.61078 ms
Game 85	35.633124 ms
Game 86	10.211757 ms
Game 87	115.535395 ms
Game 88	13.844376 ms
Game 89	2323.685826 ms
Game 90	2379.568429 ms
Game 91	447.783042 ms
Game 92	5649.308915 ms
Game 93	268.755977 ms
Game 94	192.249596 ms
Game 95	1006.087455 ms
Game 96	993.745517 ms
Game 97	301.626611 ms
Game 98	6.109814 ms
Game 99	0.019839 ms

References

[1] J. F. Groote and M. R. Mousavi, *Modeling and Analysis of Communicating Systems.* The MIT Press, 2014.

[2] M. ter Beek, E. de Vink, and T. Willemse, "Family-based model checking with mcrl2," in *Fundamental Approaches to Software Engineering* (M. Huisman and J. Rubin, eds.), Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), (Germany), pp. 387–405, Springer, 2017.

[3] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Transactions on Software Engineering*, vol. 39, pp. 1069–1089, 2013.

[4] J. Bradfield and I. Walukiewicz, *The mu-calculus and Model Checking*, pp. 871–919. Cham: Springer International Publishing, 2018.

[5] R. S. Streett and E. A. Emerson, "An automata theoretic decision procedure for the propositional mu-calculus," *Information and Computation*, vol. 81, no. 3, pp. 249 – 264, 1989.

[6] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194 – 211, 1979.

[7] I. Wegener, *Branching Programs and Binary Decision Diagrams.* Society for Industrial and Applied Mathematics, 2000.

[8] R. E. Bryant, *Binary Decision Diagrams*, pp. 191–217. Cham: Springer International Publishing, 2018.

[9] *3. Ordered Binary Decision Diagrams (OBDDs)*, pp. 45–67.

[10] 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.

[11] I. Walukiewicz, "Monadic second-order logic on tree-like structures," *Theoretical Computer Science*, vol. 275, no. 1, pp. 311 – 346, 2002.

[12] O. Friedmann and M. Lange, "Solving parity games in practice," in *Automated Technology for Verification and Analysis* (Z. Liu and A. P. Ravn, eds.), (Berlin, Heidelberg), pp. 182–196, Springer Berlin Heidelberg, 2009.

[13] F. Bruse, M. Falk, and M. Lange, "The fixpoint-iteration algorithm for parity games," *Electronic Proceedings in Theoretical Computer Science*, vol. 161, 08 2014.

[14] G. Birkhoff, *Lattice Theory.* No. v. 25,dl. 2 in American Mathematical Society colloquium publications, American Mathematical Society, 1940.

[15] A. Tarski, "A lattice-theoretical fixpoint theorem and its applications.," *Pacific J. Math.*, vol. 5, no. 2, pp. 285–309, 1955.

[16] E. A. Emerson and C. Lei, "Model checking in the propositional mu-calculus," tech. rep., Austin, TX, USA, 1986.