Verifying Featured Transition Systems using Variability Parity Games

Sjef van Loo

June 26, 2019

Contents

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).

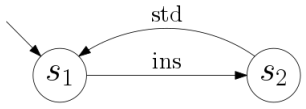


Figure 1: Coffee machine LTS C

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:*

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

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* $s \xrightarrow{a|f} s'$*.*

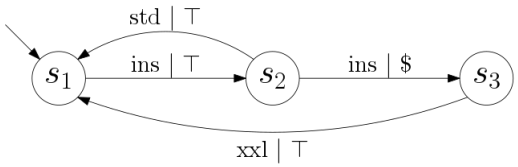


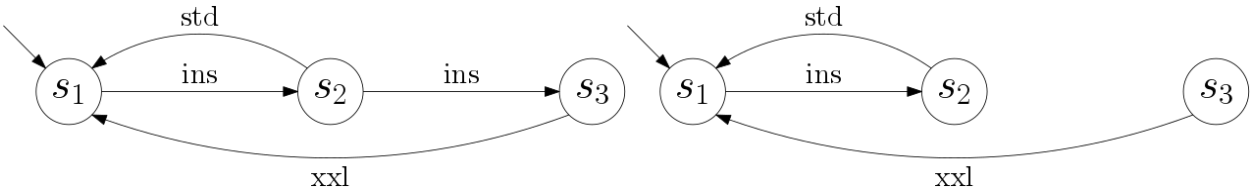
Figure 2: Coffee machine FTS C

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.

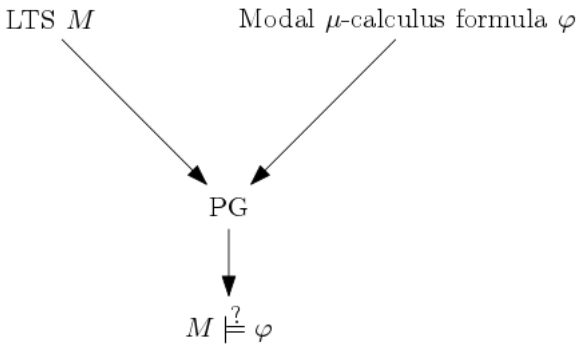


Figure 4: LTS verification using PG

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* $V_0 \cap V_1 = \emptyset$*,*
- 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.

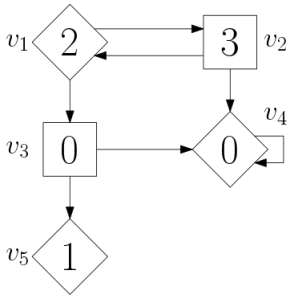


Figure 5: Parity game example

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* $\varphi_1 \vee \varphi_2 \in FL(\varphi)$ *then* $\varphi_1, \varphi_2 \in FL(\varphi)$,
- *if* $\varphi_1 \wedge \varphi_2 \in FL(\varphi)$ *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 $\varphi = \nu X.\mu Y.([ins]Y \wedge [std]X)$ 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 $\mu Y.([ins]Y \wedge [std]X)$ 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 $\psi = X$.

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

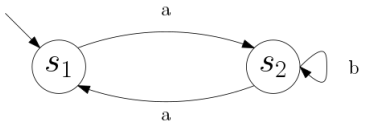


Figure 6: LTS M

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

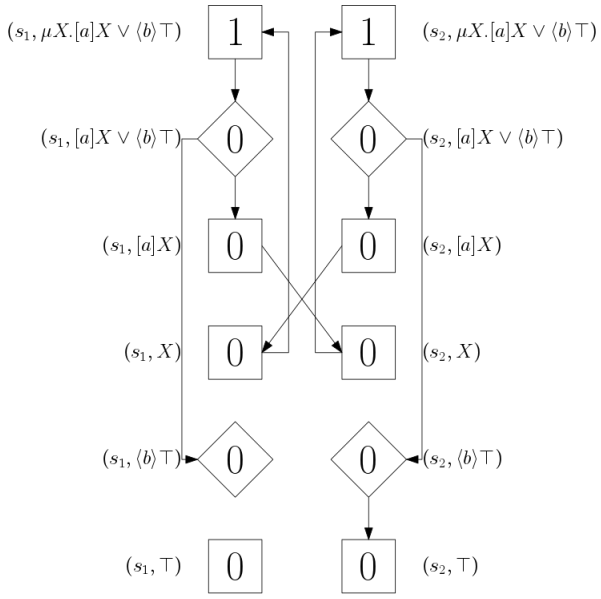


Figure 7: Parity game $LTS2PG(M, \varphi)$

resulting parity game is depicted in figure 7. Solving this parity game results in the following winning sets:

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

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

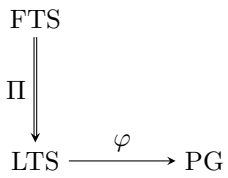
$$\begin{aligned} \sigma_0 = \{&(s_1, [a]X \vee \langle b\rangle\top) \mapsto (s_1, [a]X),\\ &(s_2, [a]X \vee \langle b\rangle\top) \mapsto (s_2, \langle b\rangle\top)\}\\ \sigma_1 = \{&\} \end{aligned}$$

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:



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

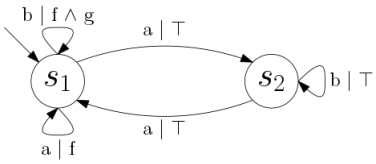


Figure 8: FTS M

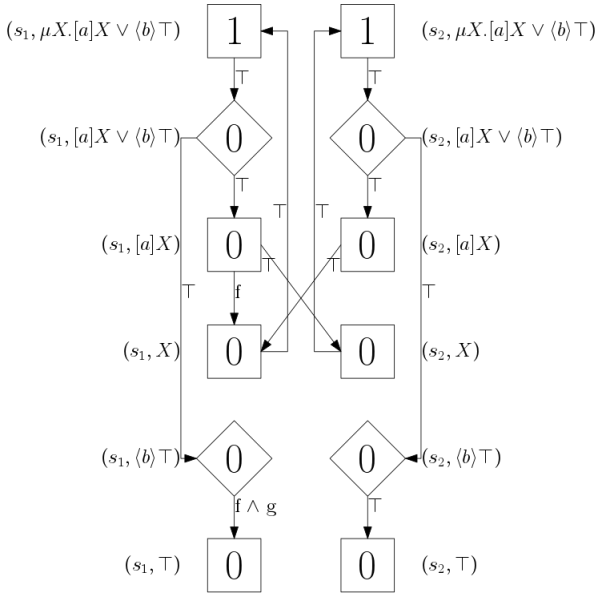


Figure 9: FPG for M and φ

$\varphi = \mu X.([a]X \vee \langle b \rangle \top)$ 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 $M_{|\{f\}}$. 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:

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

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:

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

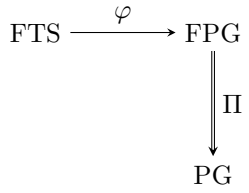
However if g is also enabled then player 0 wins in $(s_1, \langle b\rangle\top)$, thus giving the following winning sets:

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

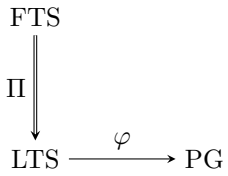
In the next section we will show how the winning sets relate to the model verification question.

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:



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:



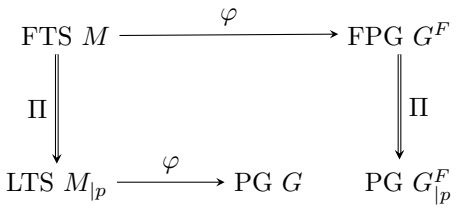
We will now show that the resulting parity games are identical.

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 $G^F_{|p} = (V^F, V_0^F, V_1^F, E^{F\prime}, \Omega^F)$, 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.



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:

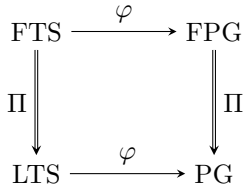
- If $\psi = \langle a \rangle \psi'$ or $\psi = [a]\psi'$ then there exists an $a \in Act$ such that $(s, a, s') \in trans'$. Using the FTS projection definition (2.5) we get $(s, a, s') \in trans$ and $p \models \gamma(s, a, s')$. Using the FTS2FPG definition (4.3) we find that $\gamma'((s, \psi), (s', \psi')) = \gamma(s, a, s')$ 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 $(s, \psi) \in V$.

□

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



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^p_α 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^p_α is equal to W_α, therefore the theorem holds. □

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*

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

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 $\{f, g\}$.

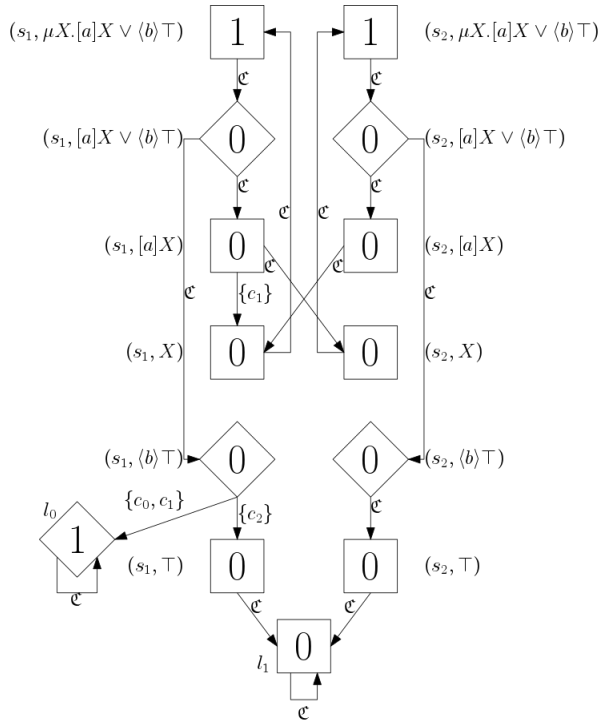


Figure 10: VPG

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 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.

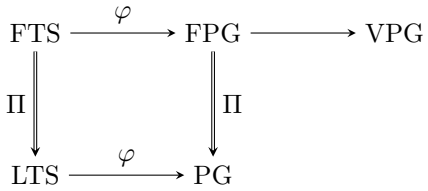
Consider infinite play π that is valid in game G for configuration p. We distinguish two cases:

- 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:



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 FPG2VPG(FTS2FPG*(M, φ)*).*

Proof. Let W_0^p and W_1^p denote the winning sets for game *FPG2VPG(FTS2FPG*(M, φ)*)*. And Q_0^p and Q_1^p denote the winning sets for game *FTS2FPG*(M, φ).

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.

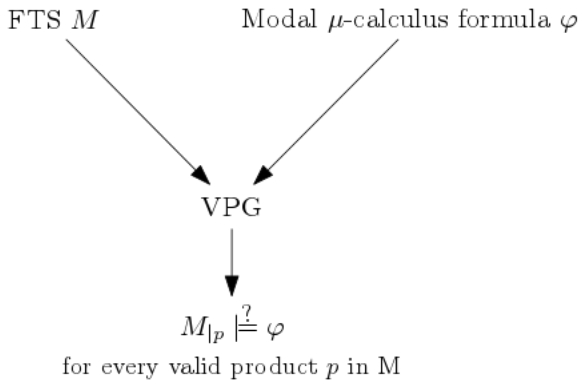


Figure 11: FTS verification using VPG

6 Solving VPGs

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 are used to improve performance.

In this next sections we explore family based algorithms, analyse their running 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.

In general we can take some existing algorithm to solve parity games with running time complexity $O(T)$ and use the algorithm to solve a VPG product based. For a VPG with configurations $\mathfrak{C}$ this gives a running time complexity of $O(|\mathfrak{C}|T)$.

6.1 Preliminaries

We explore some preliminary concepts relevant to solving algorithms.

6.1.1 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, such as the union of two sets, on sets represented as boolean functions by performing the logical and operation on the functions.

We can represent boolean functions efficiently 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 $\vee, \wedge, \neg$ 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].

7 Unified variability parity games

We can consider a VPG as a PG that is the union of all its projections. We call the resulting PG the *unification* of the VPG.

Definition 7.1. *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 trivially defined as

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

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, ie. $E \subseteq (\mathfrak{C} \times \hat{V}) \times (\mathfrak{C} \times \hat{V})$. 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 7.1. *Given VPG $\hat{G} = (\hat{V}, \hat{V}_0, \hat{V}_1, \hat{E}, \hat{\Omega}, \mathfrak{C}, \theta)$. For winning sets W_0 and W_1 for game $\hat{G}_\downarrow$ and winning sets $\hat{W}_0^c$ and $\hat{W}_1^c$ for some configuration $c \in \mathfrak{C}$ it holds that*

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

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 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 $\hat{v} \in \hat{W}_\alpha^c$. □

One of the properties of a PG is its totality; a game is total if every vertex has at least 1 outgoing vertex. Plays in a total PG will always result in an infinite path. VPGs are total, meaning that every vertex has, for every configuration $c \in \mathfrak{C}$ at least 1 outgoing vertex admitting c. Because VPGs are total a unified VPG is also total. For unified VPGs we can however further investigate its totality by introducing the notion of being total for every configuration.

Definition 7.2. *A unified VPG (V, V_0, V_1, E, Ω) is total for every configuration iff for every $(c, v) \in V$ there exists a $(c, v') \in V$ such that $((c, v), (c, v')) \in E$.*

It follows that if a unified VPG is total then it is also total for every configuration as shown in the following lemma.

Lemma 7.2. *A unified VPG $G = (V, V_0, V_1, E, \Omega)$ that is total is also total for every configuration.*

Proof. Since G is total we have for every $(c, v) \in V$ that there exists a $(c, v') \in V$ such that $((c, v), (c', v')) \in E$. Because unified VPGs don't have edges cross configurations we find that $c = c'$ and therefore G is total for every configuration. □

7.0.1 Representing unified variability parity games

Unified VPGs have a specific structure because they are the union of parity games that have the same vertices with the same owner and priority.

We can represent a set $X \subseteq (\mathfrak{C} \times \hat{V})$ 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:

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

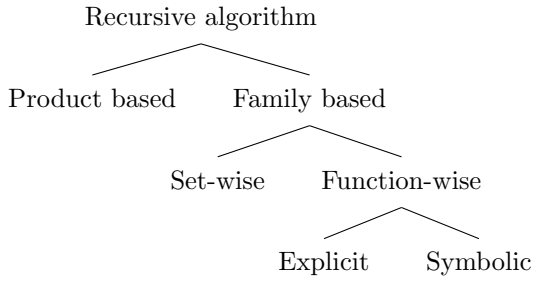
We write $\lambda^{\emptyset}$ to denote the function that maps every element to $\emptyset$, clearly $\lambda^{\emptyset} =_\lambda \emptyset$. We call using a set of pairs to represent vertices a *set-wise* representation and using functions to represent vertices a *function-wise* representation.

Finally we can simplify the priority function of a unified VPG, 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:

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

8 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 parity games. The algorithm reasons about sets of states for which certain properties hold which makes the algorithm particularly appropriate to use on unified VPGs because we can represent sets of states in unified VPGs as functions that map to sets of configurations which we can represent symbolically. This gives rise to 4 algorithms using the recursive algorithm as its basis, we depict them in the following diagram:



8.1 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.

The algorithm has a worst running time complexity of $O(e * n^d)$ where n is the number of vertices, e the number of edges and d the number of distinct priorities in the parity game. If we apply the algorithm to solve a VPG with configurations $\mathfrak{C}$ product based we get a running time of $O(|\mathfrak{C}| * e * n^d)$.

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 go from any vertex in A to a vertex in U.

Definition 8.1. *[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:

$$\alpha\text{-}Attr(G, U) = U_k$$

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)

such that for k we have

$$U_k = U_{k+1}$$

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

Definition 8.2. *[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*
- $E' = E \cap (V' \times V')$.

We can relax the subgame definition given in [10] to a definition where we only remove vertices and not remove edges.

Definition 8.3. *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 $(V', V'_0, V_1{}', {}', \Omega)$ with:*

- $V' = V \backslash U$,
- $V'_0 = V_0 \cap V'$ *and*
- $V'_1 = V_1 \cap V'$.

We will show that using the relaxed subgame definition in the recursive algorithm gives the same result.

Lemma 8.1. *Using the relaxed subgame (8.3) definition in the recursive algorithm gives the same result as using the regular subgame definition (8.2).*

Proof. Let E be the edge relation resulting from a regular subgame operation and E' be the edge relation resulting from a relaxed subgame operation.

We first observe that the difference between the two definitions is that in E only edges exists that go from and go to vertices that are in V. In E' edges can exists that go from and/or go to vertices that are not in V. We also find $E \subseteq E'$.

Next we observe that the only place where edges are used in the algorithm is in the calculation of the attractor set. We will show that the attractor set calculated for E is equal to the attractor set calculated for E'.

Let $v \in V$, we distinguish two cases:

- Case: $v \in V_\alpha$.

  Vertex v will only be attracted if there exists a $v' \in V$ such that there is an edge from v to v'. Since v and v' are both in V an edge $(v, v') \in E'$ will also exist in E, therefore the result is the same.

- Case: $v \in V_{\overline{\alpha}}$.

  Vertex v will only be attracted if all the edges that go to $v' \in V$ are in U_i. Since edges that go from v to $v'' \notin V$ are not considered the result for both E and E' will be the same.

□

8.2 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 7. Pseudo code for the modified algorithm is presented in algorithm 2.

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

Definition 8.4. *Given unified VPG $G = (V, V_0, V_1, E, \Omega)$, represented function-wise, and a non-empty set $U \subseteq V$ we define α-FAttr(G, U) such that*

$$U_0 = U$$

For $i \geq 0$:

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

Finally:

$$\alpha\text{-FAttr}(G, U) = U_k$$

such that for k we have

$$U_k = U_{k+1}$$

We will now prove that this new attractor definition gives a result equal to the original definition.

Lemma 8.2. *Given unified VPG $G = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{E}, \Omega)$ 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, V_0, V_1, E, U be the set-wise representation and $V^\lambda, V_0^\lambda, V_1^\lambda, E^\lambda, U^\lambda$ be the function-wise representation of $\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{E}, \mathcal{U}$ respectively.

The following properties hold by definition:

$$(c, \hat{v}) \in X \iff c \in X^\lambda(\hat{v}) \text{ for all } X \in \{V, V_0, V_1, U\}$$

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

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 $U_{i+1} =_\lambda U_{i+1}^\lambda$. We have $U_{i+1} =_\lambda U_{i+1}^\lambda$ iff:

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

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

Algorithm 2 RECURSIVEUVPG($PG\ G = (V : \hat{V} \to 2^{\mathfrak{C}}, V_0 : \hat{V} \to 2^{\mathfrak{C}}, V_1 : \hat{V} \to 2^{\mathfrak{C}}, E : \hat{E} \to 2^{\mathfrak{C}}, \Omega : \hat{V} \to \mathbb{N})$)

$m \leftarrow \min\{\Omega(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
$h \leftarrow \max\{\Omega(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
if $h = m$ or $V(\hat{v}) = \lambda^{\emptyset}$ **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
$U \leftarrow \bigcup\{V(\hat{v}) \mid \Omega(\hat{v}) = h\}$
$A \leftarrow \alpha$-$FAttr(G, U)$
$(W'_0, W'_1) \leftarrow$ 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
    $B \leftarrow \overline{\alpha}$-$FAttr(G, W'_{\overline{\alpha}})$
    $(W''_0, W''_1) \leftarrow$ 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)

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

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

  It follows immediately that $c \in V^\lambda_\alpha(\hat{v})$ therefore $V^\lambda_\alpha(\hat{v}) \neq \emptyset$ so we get

  $$U^\lambda_{i+1} = \bigcup_{\hat{v}'}(V^\lambda(\hat{v}) \cap E^\lambda(\hat{v}, \hat{v}') \cap U^\lambda_i(\hat{v}'))$$

  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 $c \in V^\lambda(\hat{v})$, $c \in U^\lambda_i(\hat{v}')$ and $c \in E^\lambda(\hat{v}, \hat{v}')$. If we fill this in in the above formula we can conclude that $c \in U^\lambda_{i+1}(\hat{v})$.

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

  First we observe that $(c, \hat{v}) \notin U_i$ and therefore $c \notin U^\lambda_i(\hat{v})$.

  It follows immediately that $c \in V^\lambda_\alpha(\hat{v})$ therefore $V^\lambda_\alpha(\hat{v}) \neq \emptyset$ so we get

  $$U^\lambda_{i+1} = \bigcup_{\hat{v}'}(V^\lambda(\hat{v}) \cap E^\lambda(\hat{v}, \hat{v}') \cap U^\lambda_i(\hat{v}'))$$

  Assume $c \in U^\lambda_{i+1}(\hat{v})$. There must exist a $\hat{v}'$ such that $c \in V^\lambda(\hat{v})$, $c \in E^\lambda(\hat{v}, \hat{v}')$ and $c \in U^\lambda_i(\hat{v}')$. Due to equivalence we have a vertex $(c, \hat{v}) \in V$, $((c, \hat{v}), (c, \hat{v}')) \in E$ 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: $(c, \hat{v}) \in V_{\overline{\alpha}}$ and $(c, \hat{v}) \in U_{i+1}$:
  To prove: $c \in U^\lambda_{i+1}(\hat{v})$.

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

  It follows immediately that $c \in V^\lambda_{\overline{\alpha}}(\hat{v})$ therefore $V^\lambda_{\overline{\alpha}}(\hat{v}) \neq \emptyset$ so we get

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

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

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

  which is equal to

  $$c \in E^\lambda(\hat{v}, \hat{v}') \text{ and } c \notin U^\lambda_i(\hat{v}') \text{ and } c \in V^\lambda(\hat{v}')$$

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

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

  First we observe that $(c, \hat{v}) \notin U_i$ and therefore $c \notin U^\lambda_i(\hat{v})$.

  It follows immediately that $c \in V^\lambda_{\overline{\alpha}}(\hat{v})$ therefore $V^\lambda_{\overline{\alpha}}(\hat{v}) \neq \emptyset$ so we get

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

  Since unified VPGs are total in all configurations (lemma 7.2) we have at least one outgoing edge for $(c, \hat{v})$. 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

$$c \in V^\lambda(\hat{v}') \text{ and } c \in E^\lambda(\hat{v}, \hat{v}') \text{ and } c \notin U_i^\lambda(\hat{v}')$$

Which is equal to

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

From which we conclude

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

Therefore we have $c \notin U_{i+1}^\lambda(\hat{v})$.

□

We also modified the subgame definition to work with the function-wise representation. Note that this subgame definition is similar to the relaxed subgame definition (8.3) and not to the original subgame definition (8.2).

Definition 8.5. *For unified VPG $G = (V, V_0, V_1, E, \Omega)$, represented function-wise, and set $X \subseteq V$ we define the subgame $G \backslash X = (V', V_0', V_1', E, \Omega)$ such that:*

- $V'(\hat{v}) = V(\hat{v}) \backslash X(\hat{v})$,
- $V_0'(\hat{v}) = V'(\hat{v}) \cap V_0(\hat{v})$ *and*
- $V_1'(\hat{v}) = V'(\hat{v}) \cap V_1(\hat{v})$.

We will now prove that this new subgame definition gives a result equal to the relaxed subgame definition.

Lemma 8.3. *Given unified VPG $G = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{E}, \Omega)$ and set $\mathcal{U} \subseteq \mathcal{V}$ the function-wise subgame $G \backslash \mathcal{U}$ is equal to the relaxed set-wise subgame $G \backslash \mathcal{U}$.*

Proof. Follows immediately from the subgame definitions (8.3, 8.5) and the equivalence relation between function-wise and set-wise games. □

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 8.4. *Given unified VPG $\mathcal{G} = (\mathcal{V}, \mathcal{V}_0, \mathcal{V}_1, \mathcal{E}, \Omega)$ 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 $G = (V, V_0, V_1, E, \Omega)$ be the set-wise representation of $\mathcal{G}$ and $G^\lambda = (V^\lambda, V_0^\lambda, V_1^\lambda, E^\lambda, \Omega)$ be the function-wise representation of $\mathcal{G}$.

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

Base $\mathcal{V} = \emptyset$.

RECURSIVEUVPG(G^λ) returns $\lambda^\emptyset$ and RECURSIVEPG(G) returns $\emptyset$, these two results are equal therefore the theorem holds in this case.

Base $\min\{\Omega(v) \mid v \in \mathcal{V}\} = \max\{\Omega(v) \mid v \in \mathcal{V}\}$.

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.

Set $\{v \in V \mid \Omega(v) = h\} = \{(c, \hat{v}) \in V \mid \Omega(\hat{v}) = h\} =_\lambda \bigcup\{V(\hat{v}) \mid \Omega(\hat{v}) = h\}$ by equivalence definition. So the values for U are equal in both algorithms.

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 8.2) and that the created subgames are equal (as shown in lemma 8.3). Since the subgames are equal we can apply the theorem on it by induction and conclude that the winning sets are also equal. □

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($G_\downarrow$).

8.2.1 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 it will also be considered. We prove the correctness of the algorithm in the following lemma and theorem.

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

```
1:  Queue Q ← {v̂ ∈ V̂ | A(v̂) ≠ ∅}
2:  while Q is not empty do
3:      v̂' ← Q.pop()
4:      for E(v̂, v̂') ≠ ∅ do
5:          if A(v̂) = V(v̂) then
6:              continue
7:          end if
8:          if V_α(v̂) ≠ ∅ then
9:              a ← V(v̂) ∩ E(v̂, v̂') ∩ A(v̂')
10:         else
11:             a ← V(v̂)
12:             for E(v̂, v̂'') ≠ ∅ do
13:                 a ← a ∩ (𝔠\E(v̂, v̂'') ∪ A(v̂'') ∪ 𝔠\V(v̂''))
14:             end for
15:         end if
16:         if a = ∅ then
17:             continue
18:         end if
19:         A(v̂) ← A(v̂) ∪ a
20:         Q.push(v̂)
21:     end for
22: end while
23: return A
```

Lemma 8.5. *Vertex $\hat{v}$ with 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 $V_\alpha(\hat{v}) \neq \emptyset$ then this property follows immediately from definition 8.4. If $V_{\overline{\alpha}}(\hat{v}) \neq \emptyset$ we note that parity games are total and therefore also total for every configuration (lemma 7.2), therefore vertex $\hat{v}$ has an outgoing edge to $\hat{w}$ 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 8.6. *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 $c \in \alpha$-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 in that are in A for some c are in Q so IV2 also 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.

Set A only contains vertices with configurations that are in α-FAttr(G, U). The set is only updated through lines 8-14 and 19 of the algorithm which reflects the exact definition of the attractor set therefore IV1 holds at the end of the iteration as well.

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 for any $\hat{w}$ 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 8.5). Every $\hat{w}'$ for which $A(\hat{w}')$ is updated is added to the queue (lines 16-20). Therefore there 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 case 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 8-14 and added to $A(\hat{w})$ in line 19. Therefore IV2 holds by the end of the iteration.

  Next we consider the case 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.

When the while loop ends IV1 and IV2 hold so for every 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. □

8.3 Running time

We will consider the running time for solving VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ product based, family based with a set-wise representation and family based with a function-wise representation. 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.

As stated before the product based approach using the original algorithm gives a running time of $O(c * e * n^d)$. We can also apply the original algorithm to a unified VPG (represented set-wise) for a family based approach, in this case we get a parity game with $c * n$ vertices and $O(c * e)$ edges. which gives a running time $O(c * e * (nc)^d)$.

Next we analyse the running time of the algorithm using a function-wise representation. First we consider the variant where sets of configurations are represented explicitly. We first analyse the running time of the attractor set.

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 running time for the attractor set of $O(n * c^2 * e)$.

We will now analyse the recursion of the function-wise algorithm, first we define $m = cn$ representing the total number of configuration vertex combinations. The first subgame created in the algorithm decreases d

by one, the second subgame decreases the m by one. This gives the following recursion:

$$\begin{aligned} T(m,d) &\leq T(m,d-1) + T(m-1,d) + O(n * c^2 * e) \\ T(m,1) &= O(m) \\ T(0,d) &= O(m) \end{aligned}$$

From which we get

$$\begin{aligned} T(m,d) &\leq T(m,d-1) + T(m-1,d) + O(n * c^2 * e) \\ &\leq m * T(m,d-1) + m * O(n * c^2 * e) + O(m) \\ &\leq m * T(m,d-1) + (m+1)O(n * c^2 * e) \\ T(m,1) &= O(m) \end{aligned}$$

We will now prove that $T(m,d) \leq (m+d)^d O(n * c^2 * e)$ by induction on d.

Base $d = 1$: $T(m,1) = O(m) \leq O(n * c^2 * e)$.

step

$$\begin{aligned} T(m,d) &\leq m * T(m,d-1) + (m+1)O(c^2 + n^3) \\ &\leq m * (m+d-1)^{d-1} O(n * c^2 * e) + (m+1)O(n * c^2 * e) \end{aligned}$$

Since $d > 1$ we have $m \leq m + d - 1$

$$\begin{aligned} T(m,d) &\leq (m+d-1)^d O(n * c^2 * e) + (m+1)O(n * c^2 * e) \\ &= ((m+d-1)^d + m + 1)O(n * c^2 * e) \end{aligned}$$

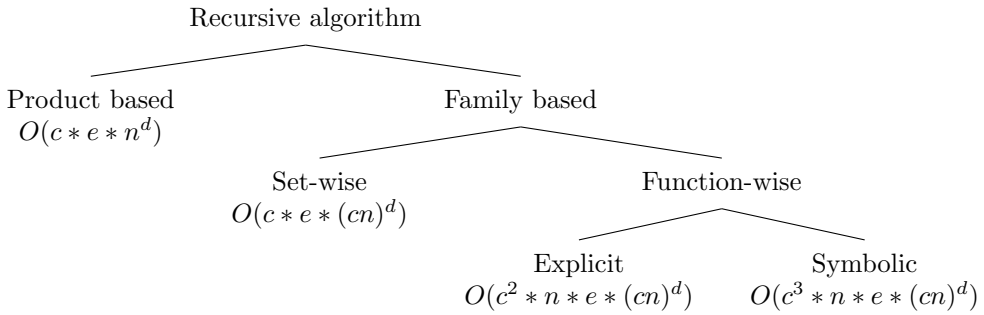
We have $(m+d-1)^d + m + 1 \leq (m+d)^d$ (using lemma A.1), we find:

$$T(m,d) \leq (m+d)^d O(n * c^2 * e)$$

We conclude that the running time for the explicit function-wise algorithm is $O((cn+d)^d n * c^2 * e) = O(c^2 * n * e * (cn)^d)$.

For the symbolic function-wise algorithm we can follow the same line of reasoning however the attractor set runs requires $O(c^2)$ to perform its set operations which gives a running time for the attractor set of $O(n * c^3 * e)$ and a total running time of $O(c^3 * n * e * (cn)^d)$.

The different algorithms, including their running times, are repeated in the diagram below:



Appendices

A Auxiliary theorems and lemma's

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

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

Proof. We expand the inequality.

$$\begin{aligned}(m+d-1)^d + m + 1 &\leq (m+d)^d \\ (m+d-1)(m+d-1)^{d-1} + m + 1 &\leq (m+d)(m+d)^{d-1} \\ m(m+d-1)^{d-1} + d(m+d-1)^{d-1} - (m+d-1)^{d-1} + m + 1 &\leq m(m+d)^{d-1} + d(m+d)^{d-1}\end{aligned}$$

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

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.