Verifying Featured Transition Systems using Variability Parity Games

Sjef van Loo

July 4, 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).

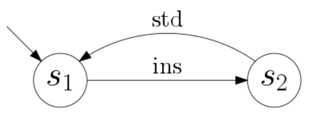


Figure 1: Coffee machine LTS C

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:*

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

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

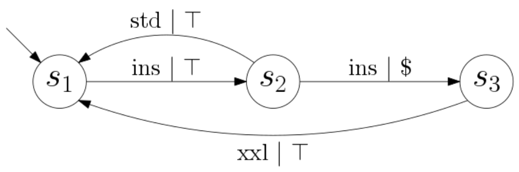


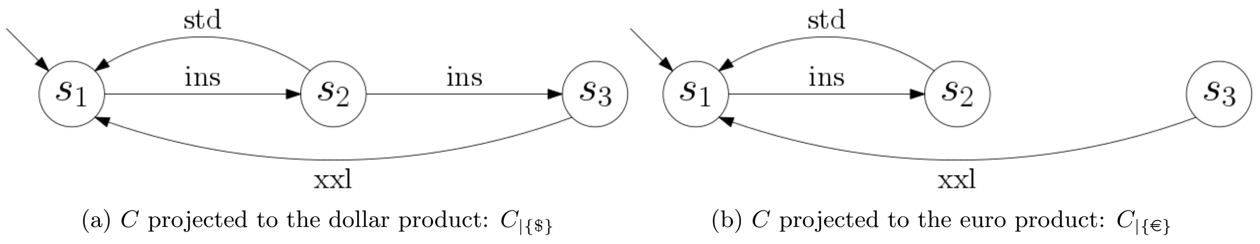
Figure 2: Coffee machine FTS C

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.

$$\begin{tikzcd}
\text{LTS } M \arrow[dr] & & \text{Modal } \mu\text{-calculus formula } \varphi \arrow[dl] \\
& \text{PG} \arrow[d] & \\
& M \overset{?}{\models} \varphi &
\end{tikzcd}$$

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.

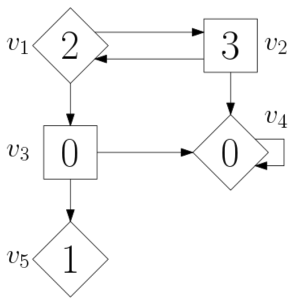


Figure 5: Parity game example

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.

- $\pi = (v_1v_2)^\omega$ is won by player 1 since the highest priority occurring infinitely often is 3.
- $\pi = v_1v_3(v_4)^\omega$ 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(φ), 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 $\psi = X$.

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

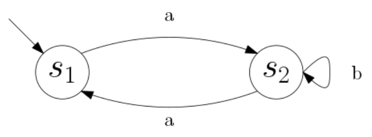
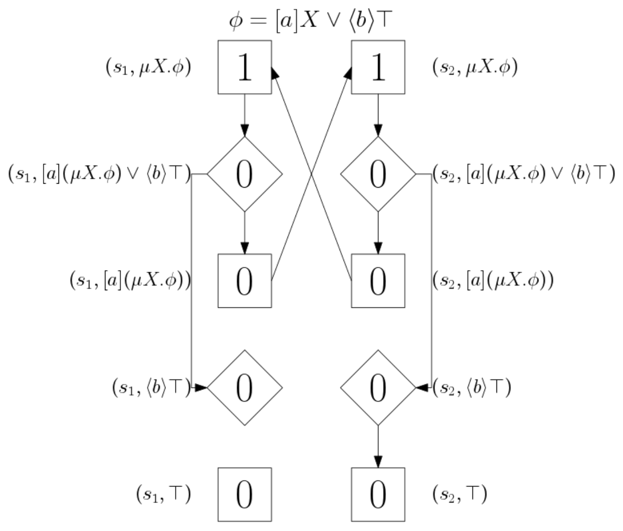


Figure 6: LTS M

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



$\phi = [a]X \vee \langle b\rangle\top$
$(s_1, \mu X.\phi)$
1
1
$(s_2, \mu X.\phi)$
$(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top)$
0
0
$(s_2, [a](\mu X.\phi) \vee \langle b\rangle\top)$
$(s_1, [a](\mu X.\phi))$
0
0
$(s_2, [a](\mu X.\phi))$
$(s_1, \langle b\rangle\top)$
0
0
$(s_2, \langle b\rangle\top)$
$(s_1, \top)$
0
0
$(s_2, \top)$

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:

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

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, φ) 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

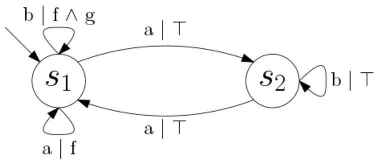


Figure 8: FTS M

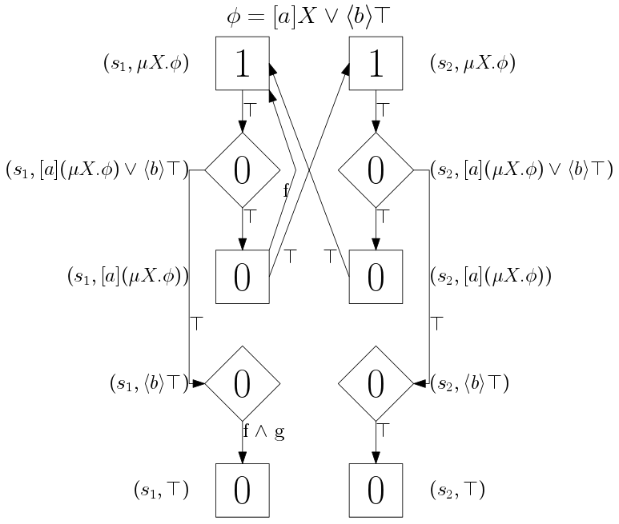

$\phi = [a]X \vee \langle b\rangle\top$
$(s_1, \mu X.\phi)$
1
1
$(s_2, \mu X.\phi)$
⊤
⊤
$(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top)$
0
0
$(s_2, [a](\mu X.\phi) \vee \langle b\rangle\top)$
f
⊤
⊤
$(s_1, [a](\mu X.\phi))$
0
⊤
⊤
0
$(s_2, [a](\mu X.\phi))$
⊤
⊤
$(s_1, \langle b\rangle\top)$
0
0
$(s_2, \langle b\rangle\top)$
f ∧ g
⊤
$(s_1, \top)$
0
0
$(s_2, \top)$

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:

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

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, φ*)*$_{|p}$ *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.

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

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 $(s, \psi) \in V$.

□

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

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

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* $FTS2FPG(M, \varphi)$.

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. □

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 $\{f, g\}$.

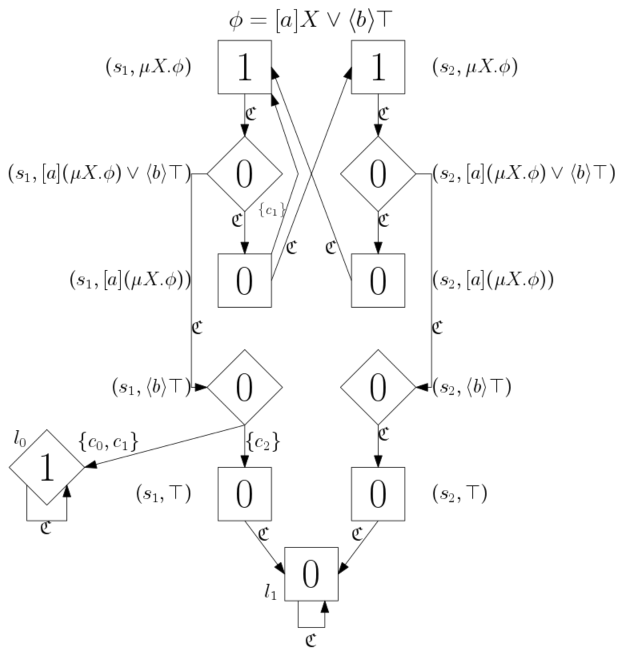

$\phi = [a]X \vee \langle b\rangle\top$
$(s_1, \mu X.\phi)$
1
1
$(s_2, \mu X.\phi)$
$\mathfrak{C}$
$(s_1, [a](\mu X.\phi) \vee \langle b\rangle\top)$
0
0
$(s_2, [a](\mu X.\phi) \vee \langle b\rangle\top)$
$\{c_1\}$
$(s_1, [a](\mu X.\phi))$
0
0
$(s_2, [a](\mu X.\phi))$
$(s_1, \langle b\rangle\top)$
0
0
$(s_2, \langle b\rangle\top)$
l_0
$\{c_0, c_1\}$
$\{c_2\}$
1
$(s_1, \top)$
0
0
$(s_2, \top)$
l_1
0

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. $\square$

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 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 $(s, \varphi) \notin W_0^p$. $\square$

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

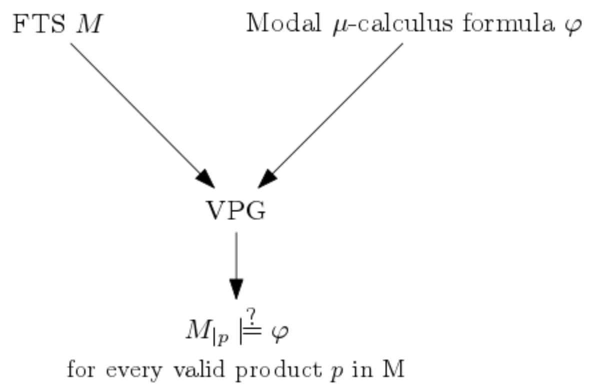

FTS M
Modal μ-calculus formula φ
VPG
$M_{|p} \overset{?}{\models} \varphi$
for every valid product p in M

Figure 11: FTS verification using VPG

6 Solving variability parity games

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.

6.1 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.

6.1.1 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 Unified parity games

We can consider create PG from a VPG by taking all the projections that are 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 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 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 7.1. *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

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

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 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 $\hat{v} \in \hat{W}_\alpha^c$. $\square$

7.0.1 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.

7.0.2 Projections and totality

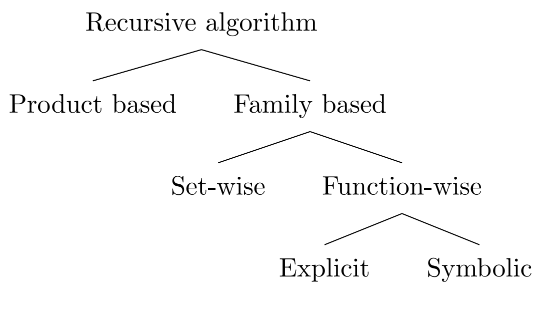
A unified PG can be projected back to one of the games from which it is the union.

Definition 7.2. *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.

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 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:



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.

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

```
1: $m \leftarrow \min\{\Omega(v) \mid v \in V\}$
2: $h \leftarrow \max\{\Omega(v) \mid v \in V\}$
3: if $h = m$ or $V = \emptyset$ then
4:     if $h$ is even or $V = \emptyset$ then
5:         return $(V, \emptyset)$
6:     else
7:         return $(\emptyset, V)$
8:     end if
9: end if
10: $\alpha \leftarrow 0$ if $h$ is even and 1 otherwise
11: $U \leftarrow \{v \in V \mid \Omega(v) = h\}$
12: $A \leftarrow \alpha\text{-}Attr(G, U)$
13: $(W'_0, W'_1) \leftarrow$ RecursivePG$(G \backslash A)$
14: if $W'_{\overline{\alpha}} = \emptyset$ then
15:     $W_\alpha \leftarrow A \cup W'_\alpha$
16:     $W_{\overline{\alpha}} \leftarrow \emptyset$
17: else
18:     $B \leftarrow \overline{\alpha}\text{-}Attr(G, W'_{\overline{\alpha}})$
19:     $(W''_0, W''_1) \leftarrow$ RecursivePG$(G \backslash B)$
20:     $W_\alpha \leftarrow W''_\alpha$
21:     $W_{\overline{\alpha}} \leftarrow W''_{\overline{\alpha}} \cup B$
22: end if
23: 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 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 $i \geq 0$:

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

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

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

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.

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

```
1: $m \leftarrow \min\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
2: $h \leftarrow \max\{\hat{\Omega}(\hat{v}) \mid V(\hat{v}) \neq \emptyset\}$
3: if $h = m$ or $V = \lambda^\emptyset$ then
4:     if $h$ is even or $V = \lambda^\emptyset$ then
5:         return $(V, \lambda^\emptyset)$
6:     else
7:         return $(\lambda^\emptyset, V)$
8:     end if
9: end if
10: $\alpha \leftarrow 0$ if $h$ is even and 1 otherwise
11: $U \leftarrow \lambda^\emptyset$, $U(\hat{v}) \leftarrow V(\hat{v})$ for all $\hat{v}$ with $\hat{\Omega}(\hat{v}) = h$
12: $A \leftarrow \alpha\text{-}FAttr(G, U)$
13: $(W_0', W_1') \leftarrow$ RECURSIVEUVPG$(G \backslash A)$
14: if $W_{\overline{\alpha}}' = \lambda^\emptyset$ then
15:     $W_\alpha \leftarrow A \cup W_\alpha'$
16:     $W_{\overline{\alpha}} \leftarrow \lambda^\emptyset$
17: else
18:     $B \leftarrow \overline{\alpha}\text{-}FAttr(G, W_{\overline{\alpha}}')$
19:     $(W_0'', W_1'') \leftarrow$ RECURSIVEUVPG$(G \backslash B)$
20:     $W_\alpha \leftarrow W_\alpha''$
21:     $W_{\overline{\alpha}} \leftarrow W_{\overline{\alpha}}'' \cup B$
22: end if
23: return $(W_0, W_1)$
```

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

Definition 8.3. *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 8.1. *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 $c \notin U_{i+1}^\lambda(\hat{v})$.

□

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

Definition 8.4. *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 8.2. *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 8.2) and $c \notin V^{\lambda'}(\hat{v})$ (by definition 8.4).

If $(c,\hat{v}) \notin U$ then $c \notin U^\lambda(\hat{v})$, also $(c,\hat{v}) \in V'$ (by definition 8.2) and $c \in V^{\lambda'}(\hat{v})$ (by definition 8.4).

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 8.2). Also $c \notin E^{\lambda'}(\hat{v},\hat{w})$ (by definition 8.4).

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 8.4). □

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 8.3. *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 8.1) and that the created subgames are equal (as shown in lemma 8.2). 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 7.1 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$(G_\downarrow)$.

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

Lemma 8.4. *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 (8.3). 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 8.5. *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$(G, A : \hat{V} \to 2^{\mathfrak{C}})$

```
Queue $Q \leftarrow \{\hat{v} \in \hat{V} \mid A(\hat{v}) \neq \emptyset\}$
while $Q$ is not empty do
    $\hat{v}' \leftarrow Q.pop()$
    for $E(\hat{v}, \hat{v}') \neq \emptyset$ do
        if $\hat{v} \in \hat{V}_\alpha$ then
            $a \leftarrow V(\hat{v}) \cap E(\hat{v}, \hat{v}') \cap A(\hat{v}')$
        else
            $a \leftarrow V(\hat{v})$
            for $E(\hat{v}, \hat{v}'') \neq \emptyset$ do
                $a \leftarrow a \cap (\mathfrak{C} \backslash E(\hat{v}, \hat{v}'') \cup A(\hat{v}''))$
            end for
        end if
        if $a \backslash A(\hat{v}) \neq \emptyset$ then
            $A(\hat{v}) \leftarrow A(\hat{v}) \cup a$
            $Q.push(\hat{v})$
        end if
    end for
end while
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 8.4). 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. □

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 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:

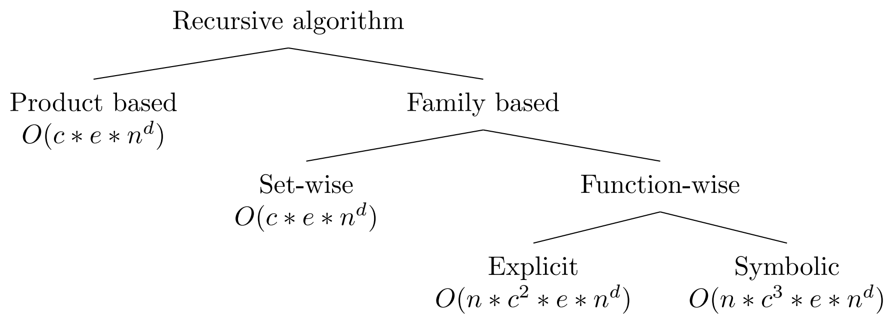
$$\begin{aligned} T(d) &\leq (n+d-1)(n+d-1)^{d-1}O(c * e) + (n+1)O(c * e) \\ &\leq (n+d-1)^d O(c * e) + (n+1)O(c * e) \\ &\leq ((n+d-1)^d + n + 1)O(c * e) \end{aligned}$$

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:



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.

9 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* VPG; 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 9.1. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$, we can create pessimistic VPG $G_{\triangleright\alpha}$ for player $\alpha \in \{0,1\}$. We have*

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

such that

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

When solving a pessimistic PG $G_{\triangleright\alpha}$ we get winning sets W_0, W_1, for which we have $W_\alpha = P_\alpha$ as shown in the following theorem.

Theorem 9.1. *Given VPG $G = (V, V_0, V_1, E, \Omega, \mathfrak{C}, \theta)$ and configuration $c \in \mathfrak{C}$ with winning sets W_0^c, W_1^c we have for any $\alpha \in \{0,1\}$ it holds that $P_\alpha \subseteq W_\alpha^c$ where P_0, P_1 are the the winning sets of game $G_{\triangleright\alpha}$.*

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. □

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.