PBES greybox implementation notes

Gijs Kant

19th January, 2019

1 Instantiation from PBES to Parity Game

1.1 PBES

Definition 1.1. *Predicate formulae* ϕ are defined by the following grammar:

$$\phi ::= b \mid \mathsf{X}(\vec{e}) \mid \neg\phi \mid \phi \oplus \phi \mid \mathsf{Q}d : D \,.\, \phi$$

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$, $\mathsf{Q} \in \{\forall, \exists\}$, b is a data term of sort Bool, $\mathsf{X} \in \mathcal{X}$ is a predicate variable, d is a data variable of sort D, and $\vec{e}$ is a vector of data terms. We will call any predicate formula without predicate variables a *simple formula*. We denote the class of predicate formulae $\mathcal{F}$.

Definition 1.2. A *First-Order Boolean Equation* is an equation of the form:

$$\sigma\mathsf{X}(d : D) = \phi$$

where $\sigma \in \{\mu, \nu\}$ is a minimum (μ) or maximum (ν) fixed point operator, d is a data variable of sort D, and ϕ is a predicate formula.

Definition 1.3. A *Parameterised Boolean Equation System (PBES)* is a sequence of First-Order Boolean Equations:

$$\mathcal{E} = (\sigma_1\mathsf{X}_1(d_1 : D_1) = \phi_1) \quad \ldots \quad (\sigma_n\mathsf{X}_n(d_n : D_n) = \phi_n)$$

We adopt the standard limitations: expressions are in positive form (negation occurs only in data expressions) and every variable occurs only once as the left hand side of an equation. A PBES that contains no quantifiers and parameters is called a *Boolean Equation System* (BES). A PBES can be *instantantiated* to a BES by expanding the quantifiers to finite conjunctions or disjunctions and substituting concrete values for the data parameters.

A one-to-one mapping can be made from a BES to an equivalent *parity game* if the BES has only expressions that are either conjunctive or disjunctive. The parity game is then represented by a game graph with nodes that represent propositional variables with concrete parameters and edges that represent dependencies. To make instantiation of a PBES to a parity game more directly we will preprocess the PBES to a format that only allows expressions to be either conjunctive or disjunctive. This format is a normal form for PBESs that we call the *Parameterised Parity Game*, defined as follows:

Definition 1.4. A PBES is a *Parameterised Parity Game* (PPG) if every right hand side of an equation is a formula of the form:

$$\mathsf{PPG} ::= \quad \bigwedge_{i\in I} f_i \wedge \bigwedge_{j\in J} \forall_{\vec{v}\in D_j} \,.\, \big(g_j \Rightarrow \mathsf{X}_j(e_j)\big) \quad \mid \quad \bigvee_{i\in I} f_i \vee \bigvee_{j\in J} \exists_{\vec{v}\in D_j} \,.\, \big(g_j \wedge \mathsf{X}_j(e_j)\big).$$

where f_i and g_j are simple boolean formulae, and e_j is a data expression. I and J are finite (possibly empty) index sets.

The expressions range over two index sets I and J. The left part is a conjunction (or disjunction) of simple expressions f_i that can be seen as conditions that should hold in the current state. The right part is a conjunction (or disjunction) of quantifiers over a (possibly empty) vector of variables for next states X_j with parameters e_j, guarded by simple expression g_j.

Before transforming arbitrary PBESs to PPG we first define another Normal Form on PBESs to make the transformation easier. This normal form can have an arbitrary sequence of bounded quantifiers as outermost operators and has a conjunctive normal form at the inner. We call this the Bounded Quantifier Normal Form (BQNF):

Definition 1.5. A First-Order Boolean formula is in *Bounded Quantifier Normal Form (BQNF)* if it has the form:

$$\begin{aligned}
\mathsf{BQNF} &::= \quad \forall \vec{d} \in D \,.\, b \Rightarrow \mathsf{BQNF} \quad | \quad \exists \vec{d} \in D \,.\, b \wedge \mathsf{BQNF} \quad | \quad \mathsf{CONJ} \\
\mathsf{CONJ} &::= \quad \bigwedge_{k \in K} f_k \wedge \bigwedge_{i \in I} \forall_{\vec{v} \in D_I} \,.\, \left(g_i \Rightarrow \mathsf{DISJ}^i\right) \\
\mathsf{DISJ}^i &::= \quad \bigvee_{\ell \in L_i} f_{i\ell} \vee \bigvee_{j \in J_i} \exists_{\vec{w} \in D_{ij}} \,.\, \left(g_{ij} \wedge \mathsf{X}_{ij}(e_{ij})\right)
\end{aligned}$$

where b, f_k, $f_{i\ell}$, g_i, and g_{ij} are simple boolean formulae, and e_{ij} is a data expression. K, I, L_i, and J_i are finite (possibly empty) index sets.

This BQNF is similar to *Predicate Formula Normal Form* (PFNF), defined elsewhere[1], in that quantification is outermost and in that the core is a conjunctive normal form. However, unlike PFNF, BQNF allows bounds on the quantified variables (hence bounded quantifiers), and universal quantification is allowed within the conjunctive part and existential quantification is allowed within the disjunctive parts. These bounds are needed to avoid problems when transforming to PPG.

1.2 Translation from BQNF to Parameterised Parity Game

In order to automatically transform a PBES to a PPG, we define a transformation function from BQNF to PPG. For brevity, we leave out the types of the parameters.
For equation system $\mathcal{E} = (\sigma \mathsf{X}_1(\vec{d}_1) = \xi_1) \quad \ldots \quad (\sigma \mathsf{X}_n(\vec{d}_n) = \xi_n)$, with each ξ_i in BQNF, the translation to PPG is defined as follows:

[1] A transformation to PFNF is implemented in the `pbesrewr` tool and documented at `http://www.win.tue.nl/mcrl2/wiki/index.php/Parameterised_Boolean_Equation_Systems`.

$$
\begin{aligned}
&s\big(\mathcal{E}\big) && \stackrel{\text{def}}{=} s\big(\sigma\mathsf{X}_1(\vec{d}_1) = \xi_1\big) \quad \ldots \quad s\big(\sigma\mathsf{X}_n(\vec{d}_n) = \xi_n\big)\\
&s\big(\sigma\mathsf{X}(\vec{d}) = f\big) && \stackrel{\text{def}}{=} \sigma\mathsf{X}(\vec{d}) = f\\
&s\big(\sigma\mathsf{X}(\vec{d}) = \forall\vec{v}\,.\,b \Rightarrow \phi\big) && \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \forall\vec{v}\,.\,b \Rightarrow t(\widetilde{\mathsf{X}}, \vec{d}+\vec{v}, \phi)\Big)\\
&&& \quad t'(\sigma, \widetilde{\mathsf{X}}, \vec{d}+\vec{v}, \phi)\\
&s\big(\sigma\mathsf{X}(\vec{d}) = \exists\vec{v}\,.\,b \wedge \phi\big) && \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \exists\vec{v}\,.\,b \wedge t(\widetilde{\mathsf{X}}, \vec{d}+\vec{v}, \phi)\Big)\\
&&& \quad t'(\sigma, \widetilde{\mathsf{X}}, \vec{d}+\vec{v}, \phi)\\
&s\big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigwedge_{k\in K} f_k\\
&\qquad \wedge \textstyle\bigwedge_{i\in I}(\forall_{\vec{v}_i}\,.\,g_i \Rightarrow \phi_i)\big) && \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigwedge_{k\in K} f_k\\
&&& \qquad\qquad \wedge \textstyle\bigwedge_{i\in I}\left(\forall_{\vec{v}_i}\,.\,g_i \Rightarrow t(\widetilde{\mathsf{X}}_i, \vec{d}+\vec{v}_i, \phi_i)\right)\Big)\\
&&& \quad t'(\sigma, \widetilde{\mathsf{X}}_1, \vec{d}+\vec{v}_1, \phi_1) \quad \ldots \quad t'(\sigma, \widetilde{\mathsf{X}}_m, \vec{d}+\vec{v}_m, \phi_m)\\
&s\big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigvee_{k\in K} f_k\\
&\qquad \vee \textstyle\bigvee_{i\in I}(\exists_{\vec{v}_i}\,.\,g_i \wedge \phi_i)\big) && \stackrel{\text{def}}{=} \Big(\sigma\mathsf{X}(\vec{d}) = \textstyle\bigvee_{k\in K} f_k\\
&&& \qquad\qquad \vee \textstyle\bigvee_{i\in I}\left(\exists_{\vec{v}_i}\,.\,g_i \wedge t(\widetilde{\mathsf{X}}_i, \vec{d}+\vec{v}_i, \phi_i)\right)\Big)\\
&&& \quad t'(\sigma, \widetilde{\mathsf{X}}_1, \vec{d}+\vec{v}_1, \phi_1) \quad \ldots \quad t'(\sigma, \widetilde{\mathsf{X}}_m, \vec{d}+\vec{v}_m, \phi_m)
\end{aligned}
$$

with $I = 1 \ldots m$, $\vec{v} \cap \vec{d} = \emptyset$ (variables in $\vec{v}$ do not occur in $\vec{d}$), b, f, f_k, g_i are simple formulae, ϕ, ϕ_i are formulae that may contain predicate variables, and

$$
t(\mathsf{X}, \vec{d}, \phi) \stackrel{\text{def}}{=} \begin{cases} \phi & \text{if } \phi = \mathsf{X}'(e),\\ \mathsf{X}(\vec{d}) & \text{otherwise;} \end{cases}
$$

$$
t'(\sigma, \mathsf{X}, \vec{d}, \phi) \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } \phi = \mathsf{X}'(e),\\ s(\sigma\mathsf{X}(\vec{d}) = \phi) & \text{otherwise.} \end{cases}
$$

1.3 Move Quantifiers Inward

Note the following equality:

$$
\forall_{\vec{d}\in D}\,.\,\bigwedge_{i\in I}\phi_i = \bigwedge_{i\in I}\left(\forall_{\vec{d}\in D}\,.\,\phi_i\right).
$$

Since the PPG form requires conjuncts of quantifiers rather than quantifiers over conjuncts, it is useful to rewrite expression such that conjunctions are more on the outside and universal quantifiers more to the inside. In the rewriting, not all parameters of the quantifier have to be

moved inward (see e.g., Example 1.6). For this we introduce the quantifier inward rewriter s_{QI}:

$$
\begin{array}{ll}
s_{QI}(b) & \stackrel{\text{def}}{=} b \\
s_{QI}(X(e)) & \stackrel{\text{def}}{=} X(e) \\
s_{QI}(\exists_{\vec{d}\in D} \cdot \phi) & \stackrel{\text{def}}{=} \exists_{\vec{d}\in D} \cdot s_{QI}(\phi) \\
s_{QI}(\bigvee_{i\in I} \phi_i) & \stackrel{\text{def}}{=} \bigvee_{i\in I} s_{QI}(\phi_i) \\
s_{QI}(\bigwedge_{i\in I} \phi_i) & \stackrel{\text{def}}{=} \bigwedge_{i\in I} s_{QI}(\phi_i) \\
s_{QI}(\forall_{\vec{d}\in D} \cdot g \implies \bigwedge_{i\in I} \phi_i) & \stackrel{\text{def}}{=} \bigwedge_{i\in I} \left(\forall_{\vec{d}\cap \mathsf{free}(\phi_i)} \cdot g_i \implies s_{QI}(\phi_i)\right)
\end{array}
$$

with ϕ an arbitrary expression in BQNF and b a data term of sort Bool and where

$$g_i = \left(\exists_{\vec{d}\cap(\mathsf{free}(g)\setminus\mathsf{free}(\phi_i))} \cdot \mathsf{filter}(g, \vec{d}\setminus \mathsf{free}(\phi_i))\right) \wedge \mathsf{filter}(g, \vec{d}\cap \mathsf{free}(\phi_i))$$

and filter is defined recursively as follows:

$$
\begin{array}{ll}
\mathsf{filter}(b, \vec{d}) & \stackrel{\text{def}}{=} \begin{cases} b & \text{if } \left(\mathsf{free}(b) \cap \vec{d}\right) = \emptyset, \\ \varnothing & \text{otherwise.} \end{cases} \\
\mathsf{filter}(\phi_1 \oplus \phi_2, \vec{d}) & \stackrel{\text{def}}{=} \begin{cases} \varnothing & \text{if } \phi_1' = \varnothing \wedge \phi_2' = \varnothing, \\ \phi_1' & \text{if } \phi_1' \neq \varnothing \wedge \phi_2' = \varnothing, \\ \phi_2' & \text{if } \phi_1' = \varnothing \wedge \phi_2' \neq \varnothing, \\ \phi_1' \oplus \phi_2' & \text{otherwise.} \end{cases}
\end{array}
$$

with $\phi_i' = \mathsf{filter}(\phi_i, \vec{d})$, $\oplus \in \{\wedge, \vee\}$ and b is a data term of sort Bool.

Example 1.6. Example transformation:

```
forall x,y . (x < 5) => ((x==a) /\ (y==b));
```

should translate to:

```
(forall x . (x < 5) => (x==a)) /\ (forall y . (exists x . x < 5) => (y==b))
```

1.4 Partitioned state vector, transition groups, and dependency matrix

We regard the instantiation of PBESs to Parity Games as generating a transition system, where states are propositional variables with concrete parameters and transitions are dependencies, specified by the right hand side of the corresponding equation in the PBES.

We use the tool LTSmin to generate a Parity Game given a PBES.

1.4.1 Partitioned state vector

A vector $\langle x_1, x_2, \ldots, x_m \rangle$ for a fixed m. *In casu* PBES instantiation, the state vector is partitioned as follows:

$$\langle \mathsf{X}, x_1, x_2, \ldots, x_k \rangle ,$$

where X is a propositional variable, and for $i \in \{1 \ldots k\}$ each x_i is the value of parameter i. k is the total number of parameter signatures in the system, ordered alphabetically; the signature consists of the name and type of the parameter. From the propositional variable X, the *type* $\in \{\wedge, \vee\}$, *priority* (an integer value) and fixpoint operator $\sigma \in \{\mu, \nu\}$ can be derived.

1.4.2 Transition groups

The equations in the PBES specify the transitions between states. These transitions can be partitioned by the part of the equation system they originate from. In this case, these are the parts of the right hand sides of the equations.

For a PBES of the form

$$\sigma \mathsf{X}(d : D) = \bigwedge_{i \in I} \forall_{\ell : D_i} \cdot g_i(d, \ell) \implies \mathsf{X}_i(h_i(d, \ell)),$$

for each $i \in I$ there is a transition group X_i which an associated transition relation $\rightarrow_{\mathsf{X}_i}$, defined as:

$$\mathsf{X}_i(d : D) \rightarrow_{\mathsf{X}_i} \mathsf{X}_i(h_i(d, \ell)),$$

for all $\ell : D_i$ such that $g_i(d, \ell)$.

Example 1.7. A specification of two sequential buffers (`buffer.2`):

$$\begin{aligned}
&\textbf{eqn} \quad N = 2;\\
&\textbf{proc } \mathsf{In}(i : \mathrm{Pos}, q : \mathrm{List}(D)) = \sum_{d:D} (\#q < N) \rightarrow \mathrm{r}_1(d) \,.\, \mathsf{In}(i, q \triangleleft d)\\
&\qquad\qquad + (q \neq []) \rightarrow \mathrm{w}(i+1, \mathrm{head}(q)) \,.\, \mathsf{In}(i, \mathrm{tail}(q));\\
&\textbf{proc } \mathsf{Out}(i : \mathrm{Pos}, q : \mathrm{List}(D)) = \sum_{d:D} (\#q < N) \rightarrow \mathrm{r}(i, d) \,.\, \mathsf{Out}(i, q \triangleleft d)\\
&\qquad\qquad + (q \neq []) \rightarrow \mathrm{s}_4(\mathrm{head}(q)) \,.\, \mathsf{Out}(i, \mathrm{tail}(q));\\
&\textbf{init} \quad \textbf{allow}(\{\mathrm{r}_1, \mathrm{c}, \mathrm{s}_4\}, \textbf{comm}(\{\mathrm{w} \mid \mathrm{r} \rightarrow \mathrm{c}\}, \mathsf{In}(1, []) \parallel \mathsf{Out}(2, [])));
\end{aligned}$$

with the property that if a message is read through r_1, it will eventually be sent through s_4:

$$[\mathbf{true}^*] \, (\forall d : D \,.\, ([\mathrm{r}_1(d)] \, (\nu \mathsf{X} \,.\, \mu \mathsf{Y} \,.\, ([\mathrm{s}_4(d)] \, \mathsf{X} \wedge [\neg \mathrm{s}_4(d)] \, \mathsf{Y}))))$$

The resulting PBES looks as follows:

$$\begin{aligned}
&\textbf{pbes } \nu \mathsf{Z}(q_{in}, q_{out} : \mathrm{List}(D)) = \ (\forall_{d:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{X}(q_{in} \triangleleft d_1, q_{out}, d)) && (1)\\
&\qquad \wedge (\forall_{d_0:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{Z}(q_{in} \triangleleft d_0, q_{out})) && (2)\\
&\qquad \wedge ((q_{out} \neq []) \Rightarrow \mathsf{Z}(q_{in}, \mathrm{tail}(q_{out}))) && (3)\\
&\qquad \wedge ((q_{in} \neq [] \wedge \#q_{out} < 2) \Rightarrow \mathsf{Z}(\mathrm{tail}(q_{in}), q_{out} \triangleleft \mathrm{head}(q_{in}))); && (4)\\
&\quad \nu \mathsf{X}(q_{in}, q_{out} : \mathrm{List}(D), d : D) = \ \mathsf{Y}(q_{in}, q_{out}, d); && (5)\\
&\quad \mu \mathsf{Y}(q_{in}, q_{out} : \mathrm{List}(D), d : D) = \ (\mathrm{head}(q_{out}) \neq d) \vee (q_{out} = []) \vee \mathsf{X}(q_{in}, \mathrm{tail}(q_{out}), d)) && (6)\\
&\qquad \wedge (\forall_{d_0:D} \,.\, (\#q_{in} < 2) \Rightarrow \mathsf{Y}(q_{in} \triangleleft d_0, q_{out}, d)) && (7)\\
&\qquad \wedge ((\mathrm{head}(q_{out}) = d) \vee (q_{out} = []) \vee \mathsf{Y}(q_{in}, \mathrm{tail}(q_{out}), d)) && (8)\\
&\qquad \wedge ((q_{in} \neq [] \wedge \#q_{out} < 2) \Rightarrow \mathsf{Y}(\mathrm{tail}(q_{in}), q_{out} \triangleleft \mathrm{head}(q_{in}), d)); && (9)\\
&\textbf{init } \mathsf{Z}([], []);
\end{aligned}$$

For this equation system, the structure of the state vector is $\langle \mathsf{X}, q_{in}, q_{out}, d \rangle$. The initial state would be encoded as $\langle \mathsf{Z}, [], [], 0 \rangle$. Since the initial state has no parameter d, a default value is chosen. The numbers 1–9 behind the equation parts denote the different transition groups, i.e., each conjunct of a conjunctive expression forms a group. E.g., $\mathsf{var}(3) = \mathsf{Z}$, $\mathsf{params}(\mathsf{var}(3)) = \langle q_{in}, q_{out} \rangle$ and $\mathsf{expr}(3) = ((q_{out} \neq []) \Rightarrow \mathsf{Z}(q_{in}, \mathrm{tail}(q_{out}))$. GROUP-NEXT$(\mathsf{Z}([], []), 3)$ yields the empty set because $q_{out} = []$. GROUP-NEXT$(\mathsf{Z}([], []), 2)$ results in $\{\mathsf{Z}([d_1], []), \mathsf{Z}([d_2], [])\}$.

1.4.3 Dependency matrix

For an equation $\sigma\mathsf{X}(d : D) = \phi$, the list of parameters is $\mathsf{params}(\mathsf{X}) \stackrel{\text{def}}{=} d : D$. Let $\mathsf{free}(d)$ be the set of *free data variables* occurring in a data term d. Let $\mathsf{used}(\phi)$ be the set of free data variables occurring in an expression ϕ such that the variables are not merely passed on to the next state. E.g., with $\mathsf{X}(a, b) = \xi$, for the expression $\phi = a \wedge \mathsf{X}(c, b)$, $\mathsf{used}(\phi) = \{a, c\}$. b is not in the set because it does not influence the computation, but is only passed on to the next state. For a formula ϕ, the function $\mathsf{changed}(\phi)$ computes the variable parameters changed in the formula:

$$\mathsf{changed}(\mathsf{X}(d_1, \ldots, d_m)) \stackrel{\text{def}}{=} \{p_i \mid i \in \{1 \ldots m\} \wedge p_i = \mathsf{params}(\mathsf{X})_i \wedge d_i \neq p_i\}$$

The function $\mathsf{tf}(\phi)$ determines if ϕ contains a branch that directly results in a **true** or **false** (not a variable). For group g and part i, we define read dependence d_R and write dependence d_W:

$$d_R(g, i) \stackrel{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } i = 1; \\ p_i \in (\mathsf{params}(\mathsf{var}(g)) \cap \mathsf{used}(\mathsf{expr}(g))) & \text{otherwise.} \end{cases}$$

$$d_W(g, i) \stackrel{\text{def}}{=} \begin{cases} (\mathsf{occ}(\mathsf{expr}(g)) \setminus \{\mathsf{var}(g)\} \neq \emptyset) \ \vee \ \mathsf{tf}(\mathsf{expr}(g)) & \text{if } i = 1; \\ p_i \in \mathsf{changed}(\mathsf{expr}(g), \emptyset) & \text{otherwise.} \end{cases}$$

Definition 1.8 (PPG Dependency matrix)**.** For a PPG P the dependency matrix $DM(P)$ is a $K \times M$ matrix defined for $1 \leq g \leq K$ and $1 \leq i \leq M$ as:

$$DM(P)_{g,i} = \begin{cases} + & \text{if } d_R(g, i) \wedge d_W(g, i); \\ r & \text{if } d_R(g, i) \wedge \neg d_W(g, i); \\ w & \text{if } \neg d_R(g, i) \wedge d_W(g, i); \\ - & \text{otherwise.} \end{cases}$$

Example 1.9. For the PBES in Example 1.7, the dependency matrix looks like this:

g	X	q_{in}	q_{out}	d
1	+	+	-	w
2	+	+	-	-
3	+	-	+	-
4	+	+	+	-
5	+	-	-	-
6	+	-	+	r
7	+	+	-	-
8	+	-	+	r
9	+	+	+	-

The first row lists the state vector parts. The left column lists the group numbers. A '+' denotes both read and write dependency, 'w' denotes write dependency, 'r' read dependency, and '-' no dependency between the group and the state vector part. The effect of caching due to this matrix can be explained by row number 5. Transition group number 5 only moves states from X to Y without affecting the parameters. Once such a transition has been computed (by GROUP-NEXT) it can be easily seen that the transition can be applied to any X-state by replacing the X with Y.

Helpful functions For an equation $\sigma\mathsf{X}(d : D) = \phi$,

$$\mathsf{params}(\mathsf{X}) \stackrel{\text{def}}{=} d : D \ .$$

Let $\mathsf{free}(d)$ be the set of *free data variables* occurring in a data term d. The function used is defined using:

$$
\begin{array}{lcl}
\mathsf{used}(d) & \stackrel{\text{def}}{=} & \mathsf{free}(d) \\
\mathsf{used}(\mathsf{X}(e)) & \stackrel{\text{def}}{=} & \ldots \text{(parameters that are used/read, not only passed on)} \\
\mathsf{used}(\phi_1 \oplus \phi_2) & \stackrel{\text{def}}{=} & \mathsf{used}(\phi_1) \cup \mathsf{used}(\phi_2) \\
\mathsf{used}(\mathsf{Q}d : D \,.\, \phi) & \stackrel{\text{def}}{=} & \mathsf{used}(\phi) \setminus \mathsf{free}(d)
\end{array}
$$

For a formula ϕ, the function $\mathsf{changed}(\phi, \emptyset)$ computes the variable parameters changed in the formula, defined as follows:

$$
\begin{array}{lcl}
\mathsf{changed}(b, L) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{changed}(\neg\phi, L) & \stackrel{\text{def}}{=} & \mathsf{changed}(\phi, L) \\
\mathsf{changed}(\phi_1 \oplus \phi_2, L) & \stackrel{\text{def}}{=} & \mathsf{changed}(\phi_1, L) \cup \mathsf{changed}(\phi_2, L) \\
\mathsf{changed}(\mathsf{Q}d : D \,.\, \phi, L) & \stackrel{\text{def}}{=} & \mathsf{changed}(\phi, L \cup \{d\}) \\
\mathsf{changed}(\mathsf{X}(d_1, \ldots, d_m), L) & \stackrel{\text{def}}{=} & \{p_i \mid i \in \{1 \ldots m\} \wedge p_i = \mathsf{params}(\mathsf{X})_i \wedge (d_i \neq p_i \vee d_i \in L)\}
\end{array}
$$

For a formula ϕ, the function $\mathsf{reset}(\phi, \vec{d})$ computes the variable parameters in $\vec{d}$ that are reset in the formula (meaning that in a successor state that parameter value will not be used), defined as follows:

$$
\begin{array}{lcl}
\mathsf{reset}(b, \vec{d}) & \stackrel{\text{def}}{=} & \emptyset \\
\mathsf{reset}(\neg\phi, \vec{d}) & \stackrel{\text{def}}{=} & \mathsf{reset}(\phi, \vec{d}) \\
\mathsf{reset}(\phi_1 \oplus \phi_2, \vec{d}) & \stackrel{\text{def}}{=} & \mathsf{reset}(\phi_1, \vec{d}) \cup \mathsf{reset}(\phi_2, \vec{d}) \\
\mathsf{reset}(\mathsf{Q}v : V \,.\, \phi, \vec{d}) & \stackrel{\text{def}}{=} & \mathsf{reset}(\phi, \vec{d}) \\
\mathsf{reset}(\mathsf{X}(e), \vec{d}) & \stackrel{\text{def}}{=} & \vec{d} \setminus \mathsf{params}(\mathsf{X})
\end{array}
$$

For a formula ϕ, the function $\mathsf{tf}(\phi)$ determines if contains a branch that directly results in a boolean value (not a variable), defined as follows:

$$
\begin{array}{lcl}
\mathsf{tf}(b) & \stackrel{\text{def}}{=} & \mathsf{true} \\
\mathsf{tf}(\neg\phi) & \stackrel{\text{def}}{=} & \mathsf{tf}(\phi) \\
\mathsf{tf}(\phi_1 \oplus \phi_2) & \stackrel{\text{def}}{=} & \mathsf{tf}(\phi_1) \vee \mathsf{tf}(\phi_2) \\
\mathsf{tf}(\mathsf{Q}d : D \,.\, \phi) & \stackrel{\text{def}}{=} & \mathsf{tf}(\phi) \\
\mathsf{tf}(\mathsf{X}(e)) & \stackrel{\text{def}}{=} & \mathsf{false}
\end{array}
$$