PBES Implementation Notes

Wieger Wesselink

April 26, 2024

This document contains details about data structures and algorithms of the PBES Library of the mCRL2 toolset.

1 Definitions

Parameterised Boolean Equation Systems (PBESs) are empty (denoted ϵ) or finite sequences of fixed point equations, where each equation is of the form $(\mu X(d{:}D) = \phi$ or $(\nu X(d{:}D) = \phi$. The left-hand side of each equation consists of a *fixed point symbol*, where μ indicates a least and ν a greatest fixed point, and a sorted predicate variable X of sort $D \to B$, taken from some countable domain of sorted predicate variables $\mathcal{X}$. The right-hand side of each equation is a predicate formula as defined below.

Definition 1 Predicate formulae ϕ *are defined by the following grammar:*

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

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$*,* $\mathsf{Q} \in \{\forall, \exists\}$*,* b *is a data term of sort* B*,* X *is a predicate variable,* d *is a data variable of sort* D *and* e *is a vector of data terms.*

The set of predicate variables that occur in a predicate formula ϕ, denoted by occ, is defined recursively as follows, for any formulae ϕ_1, ϕ_2:

$$\begin{array}{llll}
\mathsf{occ}(b) & =_{def} \emptyset & \mathsf{occ}(X(e)) & =_{def} \{X\} \\
\mathsf{occ}(\phi_1 \oplus \phi_2) & =_{def} \mathsf{occ}(\phi_1) \cup \mathsf{occ}(\phi_2) & \mathsf{occ}(\mathsf{Q}d : D.\ \phi_1) & =_{def} \mathsf{occ}(\phi_1).
\end{array}$$

Extended to equation systems, $\mathsf{occ}(\mathcal{E})$ is the union of all variables occurring at the right-hand side of equations in $\mathcal{E}$. Likewise, the set of predicate variable instantiations that occur in a predicate formula ϕ is denoted by iocc, and is defined recursively as follows

$$\begin{array}{llll}
\mathsf{iocc}(b) & =_{def} \emptyset & \mathsf{iocc}(X(e)) & =_{def} \{X(e)\} \\
\mathsf{iocc}(\phi_1 \oplus \phi_2) & =_{def} \mathsf{iocc}(\phi_1) \cup \mathsf{iocc}(\phi_2) & \mathsf{iocc}(\mathsf{Q}d : D.\ \phi_1) & =_{def} \mathsf{iocc}(\phi_1).
\end{array}$$

For any equation system $\mathcal{E}$, the set of *binding predicate variables*, $\mathsf{bnd}(\mathcal{E})$, is the set of variables occurring at the left-hand side of some equation in $\mathcal{E}$. Formally, we define:

$$\begin{array}{llll}
\mathsf{bnd}(\epsilon) & =_{def} \emptyset & \mathsf{bnd}((\sigma X(d{:}D) = \phi)\ \mathcal{E}) & =_{def} \mathsf{bnd}(\mathcal{E}) \cup \{X\} \\
\mathsf{occ}(\epsilon) & =_{def} \emptyset & \mathsf{occ}((\sigma X(d{:}D) = \phi)\ \mathcal{E}) & =_{def} \mathsf{occ}(\mathcal{E}) \cup \mathsf{occ}(\phi).
\end{array}$$

Let $\mathsf{dvar}(d)$ be the set of *free data variables* occurring in a data term d. The function dvar is extended to predicate formulae using

$$\begin{array}{llll}
\mathsf{dvar}(X(e)) & =_{def} \mathsf{dvar}(e) & \mathsf{dvar}(\mathsf{Q}d : D.\ \phi_1) & =_{def} \mathsf{dvar}(\phi_1) \setminus \mathsf{dvar}(d). \\
\mathsf{dvar}(\phi_1 \oplus \phi_2) & =_{def} \mathsf{dvar}(\phi_1) \cup \mathsf{iocc}(\phi_2). & &
\end{array}$$

The set of freely occurring predicate variables in $\mathcal{E}$, denoted $\mathsf{pvar}(\mathcal{E})$ is defined as $\mathsf{occ}(\mathcal{E}) \setminus \mathsf{bnd}(\mathcal{E})$. An equation system $\mathcal{E}$ is said to be *well-formed* iff every binding predicate variable occurs at the left-hand side of precisely one equation of $\mathcal{E}$. We only consider well-formed equation systems in this paper.

An equation system $\mathcal{E}$ is called *closed* if $\mathsf{pvar}(\mathcal{E}) = \emptyset$ and *open* otherwise. An equation $(\sigma X(d : D) = \phi)$, where σ denotes either the fixed point sign μ or ν, is called *data-closed* if the set of data variables that occur freely in ϕ is contained in the set of variables induced by the vector of variables d. An equation system is called *data-closed* iff each of its equations is data-closed.

Definition 2 Action formulae α *are defined by the following grammar:*

$$\alpha ::= b \mid \neg\alpha \mid \alpha \oplus \alpha \mid \mathsf{Q}d : D.\alpha \mid a(d) \mid \alpha\text{‘}t$$

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$*,* $\mathsf{Q} \in \{\forall, \exists\}$*,* b *is a data term of sort* B*,* X *is a predicate variable,* d *is a data variable of sort* D *and* a *is an action label.*

Definition 3 State formulae ϕ *are defined by the following grammar:*

$$\phi ::= b \mid X(e) \mid \neg\phi \mid \phi \oplus \phi \mid \mathsf{Q}d : D.\ \phi \mid \langle\alpha\rangle\phi \mid [\alpha]\phi \mid \Delta \mid \Delta(t) \mid \nabla \mid \nabla(t) \mid \sigma X(d{:}D := e)$$

where $\oplus \in \{\wedge, \vee, \Rightarrow\}$*,* $\mathsf{Q} \in \{\forall, \exists\}$*,* $\sigma \in \{\mu, \nu\}$*,* b *is a data term of sort* B*,* X *is a predicate variable,* d *is a data variable of sort* D *and* e *is a vector of data terms and* α *is an action formula.*

1.1 Well typedness constraints

1.1.1 well typedness constraints for PBES equations

- the binding variable parameters have unique names
- the names of the quantifier variables in the equation are disjoint with the binding variable parameter names
- within the scope of a quantifier variable in the formula, no other quantifier variables with the same name may occur

1.1.2 well typedness constraints for PBESs

- the sorts occurring in the global variables of the equations are declared in the data specification
- the sorts occurring in the binding variable parameters are declared in the data specification
- the sorts occurring in the quantifier variables of the equations are declared in the data specification
- the binding variables of the equations have unique names (well formedness)
- the global variables occurring in the equations are declared in the global variable specification
- the global variables occurring in the equations with the same name are identical
- the declared global variables and the quantifier variables occurring in the equations have different names
- the predicate variable instantiations occurring in the equations match with their declarations
- the predicate variable instantiation occurring in the initial state matches with the declaration
- the data specification is well typed

1.2 Monotonicity

Definition 4 *A state formula is called* monotonous *if it can be rewritten such that propositional variables are not inside the scope of a negation or an implication. More formally, a state formula* φ *is monotonous if* $m(\varphi) = true$*, where* m *is defined as follows. This definition applies to predicate formulae as well.*

$$\begin{array}{lll}
m(\neg b) & =_{def} & \mathsf{true} \\
m(\neg\neg\varphi) & =_{def} & m(\varphi) \\
m(\neg(\varphi \wedge \psi)) & =_{def} & m(\neg\varphi) \wedge m(\neg\psi) \\
m(\neg(\varphi \vee \psi)) & =_{def} & m(\neg\varphi) \wedge m(\neg\psi) \\
m(\neg(\varphi \Rightarrow \psi)) & =_{def} & m(\varphi) \wedge m(\neg\psi) \\
m(\neg\forall d{:}D.\varphi) & =_{def} & m(\neg\varphi) \\
m(\neg\exists d{:}D.\varphi) & =_{def} & m(\neg\varphi) \\
m(\neg[\alpha]\varphi) & =_{def} & m(\neg\varphi) \\
m(\neg\langle\alpha\rangle\varphi) & =_{def} & m(\neg\varphi) \\
m(\neg\nabla) & =_{def} & \mathsf{true} \\
m(\neg\nabla(t)) & =_{def} & \mathsf{true} \\
m(\neg\Delta) & =_{def} & \mathsf{true} \\
m(\neg\Delta(t)) & =_{def} & \mathsf{true} \\
m(\neg X(e)) & =_{def} & \mathsf{false} \\
m(\neg\mu X(d{:}D := e).\ \varphi) & =_{def} & m(\neg\varphi[X := \neg X]) \\
m(\neg\nu X(d{:}D := e).\ \varphi) & =_{def} & m(\neg\varphi[X := \neg X]) \\
m(b) & =_{def} & \mathsf{true} \\
m(\varphi \wedge \psi) & =_{def} & m(\varphi) \wedge m(\psi) \\
m(\varphi \vee \psi) & =_{def} & m(\varphi) \wedge m(\psi) \\
m(\varphi \Rightarrow \psi) & =_{def} & m(\neg\varphi) \wedge m(\psi) \\
m(\forall d{:}D.\varphi) & =_{def} & m(\varphi) \\
m(\exists d{:}D.\varphi) & =_{def} & m(\varphi) \\
m([\alpha]\varphi) & =_{def} & m(\varphi) \\
m(\langle\alpha\rangle\varphi) & =_{def} & m(\varphi) \\
m(\nabla) & =_{def} & \mathsf{true} \\
m(\nabla(t)) & =_{def} & \mathsf{true} \\
m(\Delta) & =_{def} & \mathsf{true} \\
m(\Delta(t)) & =_{def} & \mathsf{true} \\
m(X(e)) & =_{def} & \mathsf{true} \\
m(\mu X(d{:}D := e).\ \varphi) & =_{def} & m(\varphi) \\
m(\nu X(d{:}D := e).\ \varphi) & =_{def} & m(\varphi)
\end{array}$$

1.3 Normalization

The normalization function h is a function that eliminates implications from a state formula φ, and that 'pushes' negations inwards to the level of data expressions. A precondition of h is that φ is monotonous. If this is not the case, during the computation a term $\neg X(e)$ will be encountered.

$$\begin{array}{lll}
h(\neg b) & =_{def} & \neg b \\
h(\neg\neg\varphi) & =_{def} & h(\varphi) \\
h(\neg(\varphi \wedge \psi)) & =_{def} & h(\neg\varphi) \vee h(\neg\psi) \\
h(\neg(\varphi \vee \psi)) & =_{def} & h(\neg\varphi) \wedge h(\neg\psi) \\
h(\neg(\varphi \Rightarrow \psi)) & =_{def} & h(\varphi) \wedge h(\neg\psi) \\
h(\neg\forall d{:}D.\varphi) & =_{def} & \exists d{:}D.h(\neg\varphi) \\
h(\neg\exists d{:}D.\varphi) & =_{def} & \forall d{:}D.h(\neg\varphi) \\
h(\neg[\alpha]\varphi) & =_{def} & \langle\alpha\rangle h(\neg\varphi) \\
h(\neg\langle\alpha\rangle\varphi) & =_{def} & [\alpha]h(\neg\varphi) \\
h(\neg\nabla) & =_{def} & \Delta \\
h(\neg\nabla(t)) & =_{def} & \Delta(t) \\
h(\neg\Delta) & =_{def} & \nabla \\
h(\neg\Delta(t)) & =_{def} & \nabla(t) \\
h(\neg X(e)) & =_{def} & undefined \\
h(\neg\mu X(d{:}D := e).\ \varphi) & =_{def} & \nu X(d{:}D := e).\ h(\neg\varphi[X := \neg X]) \\
h(\neg\nu X(d{:}D := e).\ \varphi) & =_{def} & \mu X(d{:}D := e).\ h(\neg\varphi[X := \neg X]) \\
h(b) & =_{def} & b \\
h(\varphi \wedge \psi) & =_{def} & h(\varphi) \wedge h(\psi) \\
h(\varphi \vee \psi) & =_{def} & h(\varphi) \vee h(\psi) \\
h(\varphi \Rightarrow \psi) & =_{def} & h(\neg\varphi) \vee h(\psi) \\
h(\mathsf{Q}d{:}D.\varphi) & =_{def} & \mathsf{Q}d{:}D.h(\varphi) \\
h([\alpha]\varphi) & =_{def} & [\alpha]h(\varphi) \\
h(\langle\alpha\rangle\varphi) & =_{def} & \langle\alpha\rangle h(\varphi) \\
h(\nabla) & =_{def} & \nabla \\
h(\nabla(t)) & =_{def} & \nabla(t) \\
h(\Delta) & =_{def} & \Delta \\
h(\Delta(t)) & =_{def} & \Delta(t) \\
h(X(d)) & =_{def} & X(d) \\
h(\sigma X(d{:}D := e).\ \varphi) & =_{def} & \sigma X(d{:}D := e).\ h(\varphi)
\end{array}$$

2 Parity game generator

Let $\mathcal{E} = (\sigma_1 X_1(d_1 : D_1) = \varphi_1) \dots (\sigma_n X_n(d_n : D_n) = \varphi_n)$ be a PBES with initial state $X_{init}(e_{init})$, and let $R : \mathit{PbesTerm} \rightarrow \mathit{PbesTerm}$ be a rewriter. The PBES must be in normal form, i.e. it may not contain negations or implications. The following algorithm computes a BES. The generated equations are in a restricted format, such that the BES can be taken as input for a parity game solver.

```
GenerateBES($\mathcal{E}$, $X_{init}(e_{init})$, $R$)
result := $\{(\nu Y_\top = Y_\top), (\mu Y_\bot = Y_\bot)\}$
visited := $\{R(X_{init}(e_{init}))\}$
explored := $\{\top, \bot\}$
while visited $\neq \emptyset$ do
    choose $\psi \in$ visited $\setminus$ explored
    visited := visited $\setminus$ $\{\psi\}$
    explored := explored $\cup$ $\{\psi\}$
    if $\{\psi = X_k(e)\}$ then
        $\xi := R(\varphi_k[d_k := e])$
    else
        $\xi := \psi$
    if $\{\xi = X_k(e)\}$ then
        result := result $\cup$ $(\sigma_\psi Y_\psi = Y_\xi)$
        $\sigma_\xi := \sigma_k$
        visited := visited $\cup$ $\{\xi\}$
    else if $\{\xi = \bigwedge_{j \in J} \phi_j\}$ then
        result := result $\cup$ $(\sigma_\psi Y_\psi = \bigwedge_{j \in J} Y_{\phi_j})$
        for $j \in J$ do
            if $\{\xi = X_k(e)\}$ then $\sigma_{\phi_j} := \sigma_k$ else $\sigma_{\phi_j} := \sigma_\psi$
        visited := visited $\cup$ $\{\phi_j\}_{j \in J}$
    else if $\{\xi = \bigvee_{j \in J} \phi_j\}$ then
        result := result $\cup$ $(\sigma_\psi Y_\psi = \bigvee_{j \in J} Y_{\phi_j})$
        for $j \in J$ do
            if $\{\xi = X_k(e)\}$ then $\sigma_{\phi_j} := \sigma_k$ else $\sigma_{\phi_j} := \sigma_\psi$
        visited := visited $\cup$ $\{\phi_j\}_{j \in J}$
    else if $\{\xi = \top\}$ then
        result := result $\cup$ $(\sigma_\psi Y_\psi = Y_\top)$
    else if $\{\xi = \bot\}$ then
        result := result $\cup$ $(\sigma_\psi Y_\psi = Y_\bot)$
return result
```

In every step of the while loop the equation for Y_ψ is computed. If the right hand side of the equation for Y_ψ is a propositional variable instantiation, it is expanded into the right hand side of the corresponding PBES equation. Otherwise it is converted into a disjunction or conjunction by introducing new BES variables. The rewriter R is expected to eliminate all quantifiers, so the while loop does not contain cases for handling them. The order of the equations in the BES is significant. Therefore in the implementation instead of fixpoint symbols σ_ψ priority values are used. The BES variables Y_ψ are represented by integers.

An alternative for inserting the equations $(\nu Y_\top = Y_\top)$ and $(\mu Y_\bot = Y_\bot)$ at the beginning of the resulting BES is to replace $\mu Y_\psi = \top$ and $\nu Y_\psi = \top$ by $\nu Y_\psi = Y_\psi$, and to replace $\mu Y_\psi = \bot$ and $\nu Y_\psi = \bot$ by $\mu Y_\psi = Y_\psi$. This eliminates the need to introduce special equations for true and false.

This algorithm is implemented in the class `parity_game_generator`. The choice for ψ in the while loop is left to the user of the class.

3 Constant Parameter Detection and Elimination

Let $\mathcal{E} = (\sigma_1 X_1(d_{X_1} : D_{X_1}) = \varphi_{X_1}) \cdots (\sigma_n X_n(d_{X_n} : D_{X_n}) = \varphi_{X_n})$ be a PBES. Here, every d_{X_i} represents a vector of parameters. Furthermore, let $\hat{X}(\hat{e})$ be an initial state and let eval be an evaluator function on data expressions. We denote the i-th element of a vector x as $x[i]$. We also use mappings: for a mapping c, the image of i is denoted $c[i]$. The empty mapping is denoted with $\emptyset$ and the image of an element not present in a mapping is $\bot$. Note that $\emptyset[i] = \bot$ for all i. Then we define the algorithm PbesConstelm as follows:

```
PbesConstelm($\mathcal{E}$, $\hat{X}(\hat{e})$, $\mathsf{eval}$)
for $X \in \mathsf{bnd}(\mathcal{E})$ do $c_X := \emptyset$
$c_{\hat{X}} := \mathit{update}(c_{\hat{X}}, \mathsf{eval}(\hat{e}))$
$\mathit{todo} := \{\hat{X}\}$
while $\mathit{todo} \neq \emptyset$ do
    choose $X \in \mathit{todo}$
    $\mathit{todo} := \mathit{todo} \setminus \{X\}$
    for $Y(e) \in \mathsf{iocc}(\varphi_X)$ do
        if $\mathsf{eval}(\mathit{needs\_update}(Y(e), \varphi_X)[d_X := c_X]) \neq \mathit{false}$ then
            $c'_Y := \mathit{update}(c_Y, \mathsf{eval}(e[d_X := c_X]))$
            if $c'_Y \neq c_Y$ then
                $c_Y := c'_Y$
                $\mathit{todo} := \mathit{todo} \cup \{Y\}$
$\mathit{constant\_parameters} := \{(X, i) \mid c_X[i] \neq d_X[i]\}$
for $i := 1 \cdots n$ do $\varphi_{X_i} := \varphi_{X_i}[d_{X_i} := c_{X_i}]$
return $\mathit{constant\_parameters}$
```

where update_X is defined as follows:

$$\mathit{update}(c, e) =_{def} c', \text{ with } c'[i] = \begin{cases} \bot & \text{if } c = \emptyset \text{ and } e = [] \\ e[i] & \text{if } c = \emptyset \text{ and } e[i] \text{ is constant} \\ c[i] & \text{if } e[i] = c[i] \\ d_X[i] & \text{otherwise} \end{cases}$$

and where $\mathit{needs\_update}$ is a boolean function that determines whether an update should be performed. A safe choice for this function is the constant function true. [Simon Janssen, 2008] originally proposed an alternative based on a syntactical analysis of predicate formulae. The following is an improved version of his definitions.

Let c be defined as

$$\begin{array}{lcllcl}
c_T(c) & = & c & c_F(c) & = & \neg c \\
c_T(\neg\varphi) & = & c_F(\varphi) & c_F(\neg\varphi) & = & c_T(\varphi) \\
c_T(X(e)) & = & \mathit{true} & c_F(X(e)) & = & \mathit{true} \\
c_T(\mathsf{Q}d : D.\varphi) & = & \mathsf{Q}d : D.c_T(\varphi) & c_F(\mathsf{Q}d : D.\varphi) & = & \mathsf{Q}d : D.c_F(\varphi) \\
c_T(\varphi \wedge \psi) & = & c_T(\varphi) \wedge c_T(\psi) & c_F(\varphi \wedge \psi) & = & c_F(\varphi) \vee c_F(\psi) \\
c_T(\varphi \vee \psi) & = & c_T(\varphi) \vee c_T(\psi) & c_F(\varphi \vee \psi) & = & c_F(\varphi) \wedge c_F(\psi) \\
c_T(\varphi \Rightarrow \psi) & = & c_F(\varphi) \vee c_T(\psi) & c_F(\varphi \Rightarrow \psi) & = & c_T(\varphi) \wedge c_F(\psi)
\end{array}$$

and let the set *cond* be defined as

$$\begin{array}{lcl}
\mathit{cond}(X(e), c) & = & \emptyset \\
\mathit{cond}(X(e), Y(f)) & = & \emptyset \\
\mathit{cond}(X(e), \neg\varphi) & = & \mathit{cond}(X(e), \varphi) \\
\mathit{cond}(X(e), \varphi \wedge \psi) & = & \begin{cases} \{c_T(\psi)\} \cup \mathit{cond}(X(e), \varphi) & \text{if } X(e) \in \mathsf{iocc}(\varphi) \\ \{c_T(\varphi)\} \cup \mathit{cond}(X(e), \psi) & \text{if } X(e) \in \mathsf{iocc}(\psi) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{cond}(X(e), \varphi \vee \psi) & = & \begin{cases} \{c_F(\psi)\} \cup \mathit{cond}(X(e), \varphi) & \text{if } X(e) \in \mathsf{iocc}(\varphi) \\ \{c_F(\varphi)\} \cup \mathit{cond}(X(e), \psi) & \text{if } X(e) \in \mathsf{iocc}(\psi) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{cond}(X(e), \varphi \Rightarrow \psi) & = & \begin{cases} \{c_F(\psi)\} \cup \mathit{cond}(X(e), \varphi) & \text{if } X(e) \in \mathsf{iocc}(\varphi) \\ \{c_T(\varphi)\} \cup \mathit{cond}(X(e), \psi) & \text{if } X(e) \in \mathsf{iocc}(\psi) \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{cond}(X(e), \mathsf{Q}d : D.\varphi) & = & \begin{cases} \{c_\Gamma(\forall d : D.\varphi)\} \cup \{\exists d : D.\theta \mid \theta \in \mathit{cond}(X(e), \varphi)\} & \text{if } X(e) \in \mathsf{iocc}(\varphi) \\ \emptyset & \text{otherwise} \end{cases}
\end{array}$$

with $\mathsf{Q} \in \{\forall, \exists\}$. Then we define

$$\mathit{needs\_update}(X(e), \varphi) = \bigwedge_{c \in \mathit{cond}(X(e), \varphi)} c$$

The implementation of these three functions is integrated into one recursive traverser. The resulting condition is quadratic in the number of quantifier alternations in which scope X(e) occurs and linear in the other operators. Most PBESs stemming from model checking do not yield conditions larger than those contained in the LPS. Furthermore, this traverser is only executed once, a priori. This means that computing the conditions is relatively cheap.

4 Gauß Elimination

A predicate formula φ is defined by the following grammar:

$$\varphi ::= b | X(e) | \neg\varphi | \varphi \wedge \varphi | \varphi \vee \varphi | \varphi \rightarrow \varphi | \forall d : D.\varphi | \exists d : D.\varphi | true | false$$

where b is a data term of sort $\mathbb{B}$, X is a predicate variable, d is a data variable of sort D, e is a data term, $true$ represents $true$, and $false$ represents $false$.

Definition 5 *(Predicate Variable Substitution) Let φ, ψ be predicate formulae and X a predicate variable. Then we define $\psi[\varphi/X]$ as the result of applying the substitution $X := \varphi$ to the formula ψ. To make this more precise: suppose X is declared as $X(d : D)$, then any occurrence $X(\overline{d})$ in ψ is replaced by $\varphi[d := \overline{d}]$.*

Lemma 6 *(Substitution) Let $\mathcal{E}$ be an equation system for which $X, Y \notin bnd(\mathcal{E})$, then:*

$$(\sigma X(d : D) = \varphi)\mathcal{E}(\sigma' Y(e : E) = \psi) \equiv (\sigma X(d : D) = \varphi)[\psi/Y]\mathcal{E}(\sigma' Y(e : E) = \psi)$$

Definition 7 *(Approximation) Let φ, ψ be predicate formulae and X a predicate variable. We inductively define $\psi[\varphi/X]^k$ as follows:*

$$\begin{aligned} \psi[\varphi/X]^0 &\overset{def}{=} \varphi \\ \psi[\varphi/X]^{k+1} &\overset{def}{=} \psi[\varphi/X]^k \end{aligned}$$

Thus, $\psi[\varphi/X]^k$ represents the result of recursively substituting φ for X in ψ.

Lemma 8 *(Approximants as Solutions) Let φ be a predicate formula and $k \in \mathbb{N}$ be a natural number. Then*

$$\begin{aligned} (\mu X(d \quad &: \quad D) = \varphi[false/X]^k) \Rrightarrow (\mu X(d : D) = \varphi) \\ (\nu X(d \quad &: \quad D) = \varphi) \Rrightarrow (\nu X(d : D) = \varphi[true/X]^k) \end{aligned}$$

Lemma 9 *(Stable Approximants as Solutions) Let φ be a predicate formula and $k \in \mathbb{N}$ be a natural number. Then*

$$\begin{aligned} \textit{if } \varphi[false/X]^k \quad &\longleftrightarrow \quad \varphi[false/X]^{k+1} \textit{ then } (\mu X(d : D) = \varphi[false/X]^k) \equiv (\mu X(d : D) = \varphi) \\ \textit{if } \varphi[true/X]^k \quad &\longleftrightarrow \quad \varphi[true/X]^{k+1} \textit{ then } (\nu X(d : D) = \varphi[true/X]^k) \equiv (\nu X(d : D) = \varphi) \end{aligned}$$

4.1 Gauß Elimination Algorithm

Let $\mathcal{E}$ be an equation system of the form

$$\mathcal{E} = (\sigma_1 X_1(d_1 : D_1) = \varphi_1) \cdots (\sigma_n X_n(d_n : D_n) = \varphi_n),$$

and let r be a rewrite function that maps a pbes expression φ to an equivalent expression φ'.

Then we define

```
Gauss Elimination($\mathcal{E}, r$, solve)
$\mathcal{E}' := \varepsilon$
$i := n$
while not $i = 0$
do
    $\varphi_i :=$ solve$(\sigma_i X_i = \varphi_i)$
    $\mathcal{E}' := \mathcal{E}'(\sigma_i X_i = \varphi_i)$
    for $k = 1$ to $i - 1$ do $\varphi_k := r(\varphi_k[\varphi_i/X_i])$ od
    $i := i - 1$
od
return $\mathcal{E}'$
```

Here SolveEquation is an algorithm that solves a pbes equation, such that the resulting equation has no reference to the predicate variable in its right hand side. An example of such a solve equation algorithm is Approximate.

```
Approximate($\sigma X = \varphi$, compare)
$j := 0$
if $\sigma = \nu$ then $\psi_0 := true$ else $\psi_0 := false$
repeat
    $\psi_{j+1} := \varphi[\psi_j/X]$
    $j := j+1$
until compare($\psi_j, \psi_{j+1}$)
return $\sigma X = \psi_j$
```

Also pattern matching algorithms exist for this. The Gauss Elimination algorithm solves the equation system $\mathcal{E}$ for the predicate variable X_1. To solve the system $\mathcal{E}$ for all variables, the algorithm has to be applied repeatedly.

4.2 Solving a BES

If the equation system $\mathcal{E}$ is a BES (i.e. the predicate variables have no parameters), then the following simple approximate function can be used to solve it:

```
Approximate-BES($\sigma X = \varphi$)
if $\sigma = \nu$ then $\psi_0 := true$ else $\psi_0 := false$
return Simplify($\sigma X = \varphi[\psi_0/X]$)
```

5 Small progress measures

Let $\mathcal{E} = (\sigma_1 X_1 = \varphi_1) \ \cdots \ (\sigma_n X_n = \varphi_n)$ be a BES in standard recursive form. Let d be the number of μ-blocks appearing in $\mathcal{E}$. For each variable X_i we define a corresponding attribute $\alpha_i \in \mathbb{N}^d$, which is called the 'progress measure' of X_i. We define the algorithm SmallProgressMeasures as follows:

```
SmallProgressMeasures($\mathcal{E}$)
$V := \{X_i \mid 1 \leq i \leq n\}$
$V_{Even} := \{X_i \in V \mid \varphi_i$ is a disjunction$\}$
$V_{Odd} := \{X_i \in V \mid \varphi_i$ is not a disjunction$\}$
for $i := 1 \cdots n$ do $\alpha_i := [0, \cdots, 0]$
while $\mathsf{liftable\_variables}(V) \neq \emptyset$ do
    choose $X_i \in \mathsf{liftable\_variables}(V)$
    $\alpha_i := \begin{cases} \mathsf{min}\{\gamma \mid X_j \in \mathsf{occ}(\varphi_i) \wedge \gamma = f(X_j, \mathsf{rank}(X_i))\} & \text{if } X_i \in V_{Even} \\ \mathsf{max}\{\gamma \mid X_j \in \mathsf{occ}(\varphi_i) \wedge \gamma = f(X_j, \mathsf{rank}(X_i))\} & \text{if } X_i \in V_{Odd} \end{cases}$
return $(W_{Even}, W_{Odd})$
```

where $\mathsf{min}/\mathsf{max}$ is the minimum/maximum with respect to the lexicographical order on $\mathbb{N}^d$, and $\beta \in \mathbb{N}^d$ is defined as

$$\beta_i = \begin{cases} 0 & \text{if } i \text{ is even} \\ |\{X_j \in V \mid \mathsf{rank}(X_j) = i\}| & \text{if } i \text{ is odd} \end{cases}$$

and the function $\mathsf{inc} : \mathbb{N}^d \times \mathbb{Z} \to \mathbb{N}^d$ is defined inductively as

$$\begin{cases} \mathsf{inc}(\alpha, -1) & = & \top \\ \mathsf{inc}(\alpha, i) & = & \begin{cases} \mathsf{inc}([\alpha_0, \cdots, \alpha_{i-1}, 0, \alpha_{i+1}, \cdots, \alpha_d], i-1) & \text{if } \alpha_i = \beta_i \\ [\alpha_0, \cdots, \alpha_{i-1}, \alpha_i + 1, \alpha_{i+1}, \cdots, \alpha_d] & \text{otherwise} \end{cases} \end{cases}$$

and the function $f : \mathbb{N}^d \times \mathbb{N} \to \mathbb{N}^d$ is defined as:

$$f(X_j, m) = \begin{cases} [(\alpha_j)_1, \cdots, (\alpha_j)_m, 0, \cdots, 0] & \text{if } m \text{ is even} \\ \mathsf{inc}([(\alpha_j)_1, \cdots, (\alpha_j)_m, 0, \cdots, 0], m) & \text{if } m \text{ is odd} \end{cases}$$

and

$$\mathsf{liftable\_variables}(V) = \{X_i \in V \mid \mathsf{min}\{\alpha \mid w \in \mathsf{occ}(\varphi_i) \wedge \alpha = f(w)\} < \alpha_i\}.$$

and W_{Even} and W_{Odd} are defined as

$$\begin{cases} W_{Even} & = & \{X_i \in V \mid \alpha_i < \top\} \\ W_{Odd} & = & \{X_i \in V \mid \alpha_i = \top\} \end{cases}$$

and the function rank is defined inductively as follows:

$$\begin{cases} \mathsf{rank}(X_1) & = & \begin{cases} 0 & \text{if } \sigma_1 = \nu \\ 1 & \text{if } \sigma_1 = \mu \end{cases} \\ \mathsf{rank}(X_{i+1}) & = & \begin{cases} \mathsf{rank}(X_i) & \text{if } \sigma_{i+1} = \sigma_i \\ \mathsf{rank}(X_i) + 1 & \text{if } \sigma_{i+1} \neq \sigma_i \end{cases} \end{cases}$$

6 Stategraph

This section describes the implementation of the tool pbesstategraph.

6.1 Definitions

We denote the number of predicate variable instances occurring in a predicate formula φ by $\mathrm{npred}(\varphi)$. We assume that predicate variable instances in φ are assigned a unique natural number between 1 and $\mathrm{npred}(\varphi)$.

Definition 10 *Let φ be a predicate formula and let i be between* 1 *and* $\mathrm{npred}(\varphi)$. *The functions* $\mathrm{pred}(\varphi, i)$, $\mathrm{data}(\varphi, i)$ *and* $\mathrm{PVI}(\varphi, i)$ *are such that the predicate variable instance* $\mathrm{PVI}(\varphi, i)$ *is the i^{th} predicate variable instance in φ, syntactically present as* $\mathrm{pred}(\varphi, i)(\mathrm{data}(\varphi, i))$. *Let ψ be a predicate formula. We write $\varphi[i \to \psi]$ to indicate that the predicate variable instance at position i is replaced syntactically by ψ in φ.*

Definition 11 *Let φ be a predicate formula. We define the guard of predicate variable instantiation* $\mathrm{PVI}(\varphi, i)$ *for $i \leq$* $\mathrm{npred}(\varphi)$ *inductively as follows:*

$$
\begin{array}{lcl}
guard^i(c) & = & false \\
guard^i(Y) & = & true \\
guard^i(\forall d : D.\varphi) & = & guard^i(\varphi) \\
guard^i(\exists d : D.\varphi) & = & guard^i(\varphi) \\
guard^i(\varphi \wedge \psi) & = & \begin{cases} s(\varphi) \wedge guard^{i-\mathrm{npred}(\varphi)}(\psi) & \text{if } i > \mathrm{npred}(\varphi) \\ s(\psi) \wedge guard^i(\varphi) & \text{if } i \leq \mathrm{npred}(\varphi) \end{cases} \\
guard^i(\varphi \vee \psi) & = & \begin{cases} n(\varphi) \wedge guard^i(\psi) & \text{if } i > \mathrm{npred}(\varphi) \\ ns(\psi) \wedge guard^i(\varphi) & \text{if } i \leq \mathrm{npred}(\varphi) \end{cases}
\end{array}
$$

where

$$
\begin{array}{lcl}
s(\varphi) & = & \begin{cases} \varphi & \text{if } \mathrm{npred}(\varphi) = 0 \\ true & \text{otherwise} \end{cases} \\
ns(\varphi) & = & \begin{cases} \neg\varphi & \text{if } \mathrm{npred}(\varphi) = 0 \\ true & \text{otherwise} \end{cases}
\end{array}
$$

We define the function *sig* for computing significant variables recursively as follows:

$$
\begin{array}{lcl}
sig(b) & = & FV(b) \\
sig(\varphi \wedge \psi) & = & sig(\varphi) \cup sig(\psi) \\
sig(\varphi \vee \psi) & = & sig(\varphi) \cup sig(\psi) \\
sig(X(e)) & = & \emptyset \\
sig(\exists d : D.\varphi) & = & sig(\varphi) \setminus \{d\} \\
sig(\forall d : D.\varphi) & = & sig(\varphi) \setminus \{d\} \\
sig(\varphi \Rightarrow \psi) & = & sig(\varphi) \cup sig(\psi) \\
sig(\neg\varphi) & = & sig(\varphi)
\end{array}
$$

6.1.1 The functions source, target and copy

Let $X(d : D) = \varphi$ be a PBES equation. Let source be a function with the property that

$$
\mathrm{source}(X, i, j) = \begin{cases} e & \text{if } guard^i(\mathrm{PVI}(\varphi_X, i)) \Rightarrow d[j] \approx e \\ \bot & \text{otherwise} \end{cases}
$$

A possible heuristic for obtaining a source function is to look for positive occurrences of constraints of the form $d[j] \approx e$ in the guards; these can be used to define the source function. Let $\mathrm{sigma}(X, i)$ be the substitution defined as

$$
\mathrm{sigma}(X, i)(v) = \begin{array}{ll} e & \text{if } \mathrm{source}(X, i, j) = e \text{ for some } j \\ v & \text{otherwise} \end{array}
$$

We define the function target as follows:

$$\text{target}(X, i, j) = \begin{cases} c & \text{if } \mathit{rewrite}(\text{sigma}(X, i)(\text{PVI}(\varphi, i))[j] = c \\ \bot & \text{otherwise} \end{cases}$$

with c a constant. We define the function copy as follows:

$$\text{copy}(X, i, j) = \begin{cases} k & \text{if PVI}(\varphi, i)[k] = d[j] \\ \bot & \text{otherwise} \end{cases}$$

We define the function used as follows:

$$\text{used}(X, i, j) = d_X[j] \in FV(\mathit{guard}^i(\text{PVI}(\varphi_X, i)))$$

We define the function changed as follows:

$$\text{changed}(X, i, j) = \text{pred}(\varphi_X, i) = X \wedge d_X[j] \neq \text{data}(\varphi_X, i)[j]$$

Let $\mathit{par}(X)$ be the set of parameters of the equation corresponding to X. Let $\mathit{pos}(X, i)$ denote the i-th parameter of the equation corresponding to X.

6.2 Control flow parameters

Control flow parameters are computed in phases. First the function $LCFP$ is computed, then the function $GCFP$, and finally they are related using $\sim$.

6.2.1 LCFP computation

There are two versions of the computation of LCFP.

```
ComputeLocalControlflowParametersDefault($\mathcal{E}$)
for $X \in bnd(\mathcal{E})$ do
    for $n = 1, \cdots, |par(X)|$ do
        $LCFP(X, n) := true$
for $X \in bnd(\mathcal{E})$ do
    for $i = 1, \cdots, npred(\varphi_X)$ do
        if $pred(\varphi_X, i) = X$ then
            for $n = 1, \cdots, |par(X)|$ do
                if $source(X, i, n) = \bot \wedge \text{target}(X, i, n) = \bot \wedge \text{copy}(X, i, n) \neq n$ then
                    $LCFP(X, n) := false$
return $LCFP$
```

```
ComputeLocalControlflowParametersAlternative($\mathcal{E}$)
for $X \in bnd(\mathcal{E})$ do
    for $n = 1, \cdots, |par(X)|$ do
        $LCFP(X, n) := true$
for $X \in bnd(\mathcal{E})$ do
    for $i = 1, \cdots, npred(\varphi_X)$ do
        if $pred(\varphi_X, i) = X$ then
            for $n = 1, \cdots, |par(X)|$ do
                if $(\text{source}(X, i, n) = \bot \wedge \text{target}(X, i, n) = \bot \wedge \text{copy}(X, i, n) \neq \bot)$
                or $(\text{source}(X, i, n) \neq \bot \wedge \text{target}(X, i, n) \neq \bot \wedge \text{copy}(X, i, n) = \bot)$
                then
                    $LCFP(X, n) := false$
return $LCFP$
```

6.2.2 GCFP computation

```
ComputeGlobalControlflowParameters($\mathcal{E}, LCFP$)
for $X \in bnd(\mathcal{E})$ do
    for $n = 1, \cdots, |par(X)|$ do
        $GCFP(X, n) := LCFP(X, n)$
for $X \in bnd(\mathcal{E})$ do
    for $i = 1, \cdots, npred(\varphi_X)$ do
        let $Y = pred(\varphi_X, i)$
        if $Y \neq X$ then
            for $n = 1, \cdots, |par(X)|$ do
                if $\text{target}(X, i, n) = \bot \wedge \forall m : \text{copy}(X, i, m) \neq n$ then
                    $GCFP(X, n) := false$
return $GCFP$
```

6.2.3 Related control flow parameters

GCFP parameters can be related using the relation $\sim$. There are two versions of the computation of $\sim$.

```
ComputeRelatedGlobalControlflowParametersDefault($\mathcal{E}, GCFP$)
for $X \in bnd(\mathcal{E})$ do
    for $i = 1, \cdots, npred(\varphi_X)$ do
        let $Y = pred(\varphi_X, i)$
        for $n = 1, \cdots, |par(X)|$ do
            if $\text{copy}(X, i, n) = m \neq \perp$ then
                if $GCFP(X, n) \wedge GCFP(Y, m)$ then $(X, n) \sim (Y, m)$
```

```
ComputeRelatedGlobalControlflowParametersAlternative($\mathcal{E}, GCFP$)
for $X \in bnd(\mathcal{E})$ do
    for $i = 1, \cdots, npred(\varphi_X)$ do
        let $Y = pred(\varphi_X, i)$
        for $n = 1, \cdots, |par(X)|$ do
            if $\text{copy}(X, i, n) = m \neq \perp$ then
                if $GCFP(X, n) \wedge GCFP(Y, m) \ \wedge \text{target}(X, i, m) = \perp$ then $(X, n) \sim (Y, m)$
```

6.2.4 Control flow graphs

The symmetric closure $\tilde{}_S$ of the relation $\sim$ defines an undirected graph $(V, \tilde{}_S)$ on the set of vertices $V = \{(X, i) \mid X \in bnd(\mathcal{E}) \wedge 1 \leq i \leq |d_X|\}$. This graph is called the global control flow graph. The connected components in this graph are the local control flow graphs. A local control flow graph is called *invalid* if it contains two vertices (X, i) and (X, j) with $i \neq j$.

6.3 Global algorithm

6.3.1 Global control flow graph

The following algorithm is computes the global control flow graph.

```
ComputeGlobalControlflowGraph($\mathcal{E}$, $X_{init}(e_{init})$)
$V := \emptyset$
$E := \emptyset$
$todo := \{(X_{init}, e) \mid \forall k \leq |c_X| : e[k] = e_{init}[\downarrow_X k]\}$
while $todo \neq \emptyset$ do
    choose $u \in todo$
    $todo := todo \setminus \{u\}$
    $V := V \cup \{u\}$
    for $i = 1 \cdots \mathrm{npred}(\varphi)$ do
        if EnabledEdge($u, i$)
            $v :=$ ComputeVertex($u, i, \mathrm{PVI}(\varphi_X, i)$)
            if $v \notin V$
                $V := V \cup \{v\}$
                $todo := todo \cup \{v\}$
            $E := E \cup \{(u, i, v)\}$
return $(V, E)$
```

```
EnabledEdge($u, i$)
let $u = (X, e)$
for $k = 1 \cdots |c_X|$ do
    if $\mathrm{source}(X, i, \downarrow_X k) \neq \bot \wedge \mathrm{source}(X, i, \downarrow_X k) \neq e_k$ then
        return $false$
return $true$
```

```
ComputeVertex($u, i, Y(f)$)
let $u = (X, e)$
for $l = 1 \cdots |c_Y|$ do
    $q := \mathrm{target}(X, i, \downarrow_Y l)$
    if $q = \bot$ then
        choose $k$ such that $\mathrm{copy}(X, i, \downarrow_X k) = \downarrow_Y l$
        $v_l := e_k$
    else
        $v_l := q$
return $(Y, v)$
```

Note that $\mathrm{copy}(X, i, \downarrow_Y l) = \downarrow_X k$ implies that parameters $(X, \downarrow_X k)$ and $(Y, \downarrow_Y l)$ are related.

Remark 12 *In the code* $\downarrow_X k$ *is represented by* `cfp_X[k]`

Remark 13 *In the code* $\mathrm{copy}(X, i, \downarrow_X k)$ *is represented by* `Yf.copy(k)`*, where* $Yf = \mathrm{PVI}(\varphi_X, i)$

6.3.2 Global control flow marking

The following algorithm computes the function *marking* that denotes which parameters are marked in a vertex of the control flow graph (V, E).

```
ComputeMarkingGlobal($\mathcal{E}$, $X_{init}(e_{init})$, $V$, $E$)
for $u = X(e) \in V$ do $marking(u) := sig(u) \cap par(X)$
$todo := V$
while $todo \neq \emptyset$ do
    choose $v = X(e) \in todo$
    $todo := todo \setminus \{v\}$
    for $(u, v) \in E$ do
        let $X(f) = label(u, v)$
        for $d_X[j] \in marking(v)$
            $M := (FV(f[j]) \setminus marking(u)) \cap par(X)$
            if $M \neq \emptyset$
                $marking(u) := marking(u) \cup M$
                $todo := todo \cup \{u\}$
```

6.3.3 Global reset variables

Let PVI$(\varphi_X, i) = Y(e)$ and let V be the global control flow graph. Then we define

```
ResetVariableGlobal($Y(e)$, $i$, $V$)
$\varphi := true$
for $u = Y(f) \in V$ do
    $c := true$
    $k := 1$
    for $j = 1 \cdots |par(Y)|$ do
        $r := []$
        if $CFP(Y, j)$ then
            if $\text{target}(X, i, j) = \bot$
                $c := c \wedge (e[j] = f[k])$
            $r := r \lhd f[k]$
        else if $e[j] \in marking(u)$ then
            $r := r \lhd e[j]$
        else
            $r := r \lhd default\_value(e[j])$
        $k := k + 1$
    $\varphi := \varphi \wedge (c \Rightarrow Y(r))$
return $\varphi$
```

6.4 Compute values

Let C be a component containing related CFPs.

```
$\textsc{ComputeValues}(\mathcal{E},\ X_{init}(e_{init}), C)$
$result := \emptyset$
for $(X, j) \in C$ do
    if $(X = X_{init})$ then $result := result \cup \{e_{init}[j]\}$
for $(X, k) \in C$ do
    for $i = 1 \cdots \mathrm{npred}(\varphi_X)$ do
        if $\mathrm{source}(X, i, k) = v$ then $result := result \cup \{v\}$
for $(Y, k) \in C$ do
    for $i = 1 \cdots \mathrm{npred}(\varphi_Y)$ do
        if $\mathrm{pred}(\varphi_Y, i) = Y \wedge \mathrm{target}(Y, i, k) = v$ then $result := result \cup \{v\}$
return $result$
```

6.5 Local algorithm

6.5.1 Local control flow graph

Given a GCFG $(W, \tilde{})$ and a component $C \subseteq W$, then we define U as $\{(X, i, d = e) \mid (X, i, d) \in C \wedge e \in \textsc{ComputeValues}(C)\} \cup \{(X_{init}, ?, ? =?) \mid \forall i : (X_{init}, i) \notin C\}$. Note that this algorithm potentially extends the graph with 'undefined' nodes in a lazy fashion.

```
$\textsc{ComputeLocalControlFlowGraph}(U, C)$
$V, E := U, \emptyset$
$todo := U$
while $todo \neq \emptyset$ do
    choose $u = (X, k, d = e) \in todo$
    $todo := todo \setminus \{u\}$
    for $i = 1 \cdots \text{npred}(\varphi_X)$ do
        let $Y = \text{pred}(\varphi_X, i)$
        if $d =?$ then
            if $(Y, k') \in C$ for some $k'$ then
                if $\text{target}(X, i, k') = e'$ then
                    $v := (Y, k', d_Y[k'] = e')$
                    $insert(V, E, todo, u, i, v)$
            else
                if $X \neq Y$ then
                    $v := (Y, ?, ? =?)$
                    $insert(V, E, todo, u, i, v)$
        else
            if $(Y, k') \in C$ for some $k'$ then
                if $(\text{source}(X, i, k) = e \ \wedge \text{target}(X, i, k') = e')$ then $insert(V, E, todo, u, i, (Y, k', d_Y[k'] = e'))$
                else if $(Y \neq X \wedge \text{source}(X, i, k) = \bot \ \wedge \text{target}(X, i, k') = e')$ then $insert(V, E, todo, u, i, (Y, k', d_Y[k'] =$
                else if $(Y \neq X \wedge \text{source}(X, i, k) = \bot \ \wedge \text{copy}(X, i, k) = k')$ then $insert(V, E, todo, u, i, (Y, k', d_Y[k'] = e)$
            else
                $v := (Y, ?, ? =?)$
                $insert(V, E, todo, u, i, v)$
return $(V, E)$,
```

where $insert(V, E, todo, u, i, v)$ is shorthand for the statements

```
if $v \notin todo \wedge v \notin V$ then $todo := todo \cup \{v\}$
$V := V \cup \{v\}$
$E := E \cup \{(u, i, v)\}$
```

6.5.2 Local belongs relation

Let $(V, \longrightarrow)$ be a local control flow graph, and rules be a predicate defined as

$$\text{rules}(V, X, i) = \exists_{(X,j,e) \in V} : (X, j, e) \xrightarrow{i}.$$

```
ComputeBelongs( $V$, $\longrightarrow$, rules)
$B_k := \emptyset$
for $(X, j, v) \in V$ do
    $belongs := \{m \mid d_X[m]$ is a data parameter of $X\}$
    for $i = 1 \cdots \text{npred}(\varphi)$ do
        for $m \in belongs$ do
            if $(\text{used}(X, i, m) \vee \text{changed}(X, i, m)) \wedge \neg\text{rules}(V, X, i)$ then
                $belongs := belongs \setminus \{m\}$
    $B_k := B_k \cup \{(X, d_X[m]) \mid m \in belongs\}$
return $B_k$
```

6.5.3 Local control flow marking

Let (V, E) be a local control flow graph, and B the corresponding belongs relation.

```
UpdateMarkingRule1($B$, $u$, $i$, $v$)
let $u = (X, n, d_X[n] = z)$
let $v = (Y, m, d_Y[m] = w)$
let $Y(e) = \text{PVI}(\varphi_X, i)$
$M := \emptyset$
for $d_Y[l] \in marking(v)$ do
    $M := M \cup (\text{FV}\,(\text{rewr}(e[l], [d_X[n] := z])) \cap \{d \mid (X, d) \in B\})$
return $marking(u) \cup M$
```

```
UpdateMarkingRule2($B$, $u$, $i$, $v$)
let $u = (X, n, d_X[n] = z)$
let $v = (Y, m, d_Y[m] = w)$
let $Y(e) = \text{PVI}(\varphi_X, i)$
$M := \emptyset$
for $d_Y[l] \in marking(v)$ do
    if $(Y, d_Y[l]) \notin B$ then
        $M := M \cup (\text{FV}\,(\text{rewr}(e[l], [d_X[n] := z])) \cap \{d \mid (X, d) \in B\})$
return $marking(u) \cup M$
```

```
ComputeMarkingLocal($\mathcal{E}$, $(V_1, E_1, B_1), \cdots, (V_J, E_J, B_J)$)
for $j = 1 \cdots J$ do
    for $u = (X, n, d_X[n] = z) \in V_j$ do
        $marking(u) := significant(u) \cap \{d \mid (X, d) \in B_j\}$
$stable := false$
while $\neg stable$ do
    $stableint := false$
    while $\neg stableint$ do
        $stableint := true$
        for $j = 1 \cdots J$ do
            $todo := V_j$
            while $todo \neq \emptyset$ do
                choose $u = (X, n, d_X[n] = z) \in todo$
                $todo := todo \setminus \{u\}$
                if $marking(u) = \{d \mid (X, d) \in B_j\}$ then continue
                for $(u, i, v) \in E_j$ do
                    $m := marking(u)$
                    $marking(u) :=$ UpdateMarkingRule1($B_j$, $u$, $i$, $v$)
                    if $m \neq marking(u)$ then
                        $todo := todo \cup \{v' \mid \exists i' : (v', i', u) \in E_j\}$
                        $stableint := false$
    $stableext := false$
    while $\neg stableext$ do
        $stableext := true$
        for $j = 1 \cdots J$ do
            for $u = (X, n, d_X[n] = z) \in V_j$ do
                if $marking(u) = \{d \mid (X, d) \in B_j\}$ then continue
                for $(u, i, w) \in E_j$ do
                    let $Y(e) = \mathrm{PVI}(\varphi_X, i)$
                    for $k = 1 \cdots J$ do
                        if $k \neq j$ then
                            for $v = (Y, m, d_Y[m] = w) \in V_k$ do
                                if $\exists v' = (X, m', D_X[m']) \in V_k$ such that $(v', i, v) \in E_k$ then
                                    $m := marking(u)$
                                    $marking(u) :=$ UpdateMarkingRule2($B_j$, $u$, $i$, $v$)
                                    if $m \neq marking(u)$ then
                                        $stableint := false$
                                        $stableext := false$
    $stable := stableint \wedge stableext$
```

6.5.4 Local reset variables

```
ResetVariableLocal($i$, $\sigma X(d_X) = \varphi_X, V_1, \cdots, V_J, B_1, \cdots, B_J, \text{rules}$)
let $Y(e) = \text{PVI}(\varphi_X, i)$
$e' := e$
for $k = 1 \cdots |e|$ do
    if $\text{CFP}(Y, k)$ then continue
    $relevant := true$
    $condition := \{\}$
    for $j = 1 \cdots J$ do
        if $\text{rules}(V_j, X, i)$ then
            let $p, q$ be such that $(Y, p, q) \in V_j$
            if $\text{target}(X, i, p) \neq \perp$ then
                let $q' = \text{target}(X, i, p)$
                $relevant := relevant \wedge ((Y, d_Y[k]) \in B_j \Rightarrow d_Y[k] \in marking(Y, p, q'))$
            else
                $relevant := relevant \wedge ((Y, d_Y[k]) \in B_j \Rightarrow \exists r : d_Y[k] \in marking(Y, p, r))$
                $condition{+}= \{e[p] = r \mid (Y, d_Y[k]) \in B_j \wedge (Y, p, r) \in V_j \wedge d_Y[k] \notin marking(Y, p, r)\}$
    if $\neg relevant$ then
        $e'[k] := default\_value(d_Y[k])$
    else if $condition \neq \{\}$ then
        $e'[k] := if(join\_or(condition), default\_value(d_Y[k]), e[k]))$
return $Y(e')$
```

7 ReachableVariables

Let $\mathcal{E} = (\sigma_1 X_1(d_{X_1} : D_{X_1}) = \varphi_{X_1}) \cdots (\sigma_n X_n(d_{X_n} : D_{X_n}) = \varphi_{X_n})$ be a PBES, and let $X_{init}(e_{init})$ be the initial state. The algorithm ReachableVariables computes the reachable predicate variables.

```
ReachableVariables($\mathcal{E}, X_{init}$)
visited := $\{X_{init}\}$; explored := $\emptyset$
while visited $\neq \emptyset$
    choose $X_i \in$ visited
    visited := visited $\setminus \{X_i\}$
    explored := explored $\cup \{X_i\}$
    for each $X_j(e) \in \mathsf{iocc}(\varphi_{X_i})$
        if $X_j \notin$ explored
            visited := visited $\cup \{X_j\}$
return explored
```

8

ATerm format

< DataExpr >	c
StateTrue	$true$
StateFalse	$false$
StateNot(< StateFrm >)	$\neg\varphi$
StateAnd(< StateFrm >, < StateFrm >)	$\varphi \wedge \varphi$
StateOr(< StateFrm >, < StateFrm >)	$\varphi \vee \varphi$
StateImp(< StateFrm >, < StateFrm >)	$\varphi \Rightarrow \varphi$
StateForall(< DataVarId > +, < StateFrm >)	$\forall x{:}D.\varphi$
StateExists(< DataVarId > +, < StateFrm >)	$\exists x{:}D.\varphi$
StateMust(< RegFrm >, < StateFrm >)	$\langle\alpha\rangle\varphi$
StateMay(< RegFrm >, < StateFrm >)	$[\alpha]\varphi$
StateYaled	∇
StateYaledTimed(< DataExpr >)	$\nabla(t)$
StateDelay	Δ
StateDelayTimed(< DataExpr >)	$\Delta(t)$
StateVar(< String >, < DataExpr > ∗)	$X(d)$
StateNu(< String >, < DataVarIdInit > ∗, < StateFrm >)	$\nu X(x{:}D := d).\ \varphi$
StateMu(< String >, < DataVarIdInit > ∗, < StateFrm >)	$\mu X(x{:}D := d).\ \varphi$

Naming conventions

$\mathsf{left}(\varphi \otimes \psi)$	$=$	φ
$\mathsf{right}(\varphi \otimes \psi)$	$=$	ψ
$\arg(\neg\varphi)$	$=$	φ
$\arg(\forall d : D.\varphi) = \arg(\exists d : D.\varphi)$	$=$	φ
$\mathsf{var}(\forall d : D.\varphi) = \mathsf{var}(\exists d : D.\varphi)$	$=$	$d : D$
$\arg(\langle\alpha\rangle\, \varphi) = \arg([\alpha]\varphi)$	$=$	φ
$\mathsf{act}(\langle\alpha\rangle\, \varphi) = \mathsf{act}([\alpha]\varphi)$	$=$	α
$\mathsf{time}(\nabla(t)) = \mathsf{time}(\Delta(t))$	$=$	t
$\mathsf{var}(X(d : D))$	$=$	$d : D$
$\arg(\sigma X(d : D := e).\varphi)$	$=$	φ
$\mathsf{name}(\sigma X(d : D := e).\varphi)$	$=$	X
$\mathsf{var}(\sigma X(d : D := e).\varphi)$	$=$	$d : D$
$\mathsf{val}(\sigma X(d : D := e).\varphi)$	$=$	e

where σ is either μ or ν, and $\otimes$ is either $\wedge$, $\vee$, or $\Rightarrow$.