Eindhoven, August 5, 2008

# Tools for parameterized boolean equation systems

by

Simon Janssen

in partial fulfillment of the requirements for the degree of

**Master of Science**
**in Computer Science and Engineering**

Supervisor:
T.A.C Willemse, TU/e department of Mathematics and Computer Science
J.W. Wesselink, TU/e department of Mathematics and Computer Science

## Abstract

Boolean equation systems (BESs) are useful for verifying properties on transition systems. An extension of BESs are the parameterized boolean equation systems (PBESs). A PBES extends a normal BES with data, which increases the expressiveness of the equation system. A typical use of a PBES is verifying a modal mu-calculus formula on a possibly infinite process. Solving a PBES for the general case is undecidable, but when certain data sorts, like finite data sorts, are used, a PBES can be solved. Sometimes a PBES holds too much data, i.e. not all data influences the solution of a PBES. If this is the case the parameters containing this data can be removed. It can also happen that only the solution of a certain predicate about the PBES is required. In this case the PBES can sometimes also be reduced. This article discusses some ways to reduce PBESs.

## Acknowledgments

I would like to thank my supervisor Tim Willemese for his guidance and input throughout this project, and for all the help to make this thesis consistent.
Many thanks also go to Wieger Wesselink for his great input, feedback and of course for implementing the algorithms described in this article. Wieger helped out a lot when Tim was taking care of his newborn son.
I also would like to thank Rink Springer for reading earlier versions of this thesis and providing it with useful comments.

# Contents

# Chapter 1

# Introduction

Boolean equation systems (BESs) have been the topic of study for some time. In [Mad97] a detailed overview of properties of BESs is described. There are a number of problems that can be encoded into a BES. By solving a BES the encoded problem is solved as well. An extension of BESs are parameterized boolean equation systems (PBESs). A PBES extends a BES with data. This makes it possible to express a larger range of problems. One way a PBES can be constructed is from a process specification and a modal $\mu$-calculus formula [SW89]. Such a modal $\mu$-calculus formula states a property about the process. A PBES created from a process and a modal $\mu$-calculus formula can be solved; if the solution is *true* the used property holds on the used process, if the solution of the PBES is $false$ the property does not hold. This makes PBESs a powerful model checking tool.

The goal of the theory and algorithms presented in this thesis is to simplify PBESs by reducing the number of data parameters that occur in the equations of PBESs. Reduction of parameters is not only important because it speeds up the calculation of a PBES; solving a PBES in general is undecidable, however, if all parameters with infinite data sorts can be removed from the PBES, the PBES can always be solved. There are different ways to solve a PBES. In this thesis we will discuss the impact of the algorithms presented on two known techniques, namely instantiation, see [DPWar], and symbolic approximation, see [GW04].

Tools from the mCRL2 toolkit [GMR+07] are used to construct PBESs from mCRL2-specifications. The algorithms described in this thesis will be implemented and added to this mCRL2 toolkit. Both techniques mentioned for solving PBESs can benefit from the proposed algorithms, however we can only conduct tests for instantiation, since only this algorithm is implemented in the mCRL2 toolkit.

We start this thesis be defining the syntax and semantics of PBESs in Chapter 2. After that we introduce the algorithm $ParElm$, which detects and removes superfluous parameters from PBESs. This is done in Chapter 3. Chapter 4 introduces the algorithm $ConstElm$, which tries to replace parameters by a constant value. We finish by giving a conclusion and a word on future work in Chapter 5. A way both $ParElm$ and $ConstElm$ can be implemented is found in Appendix A and Appendix B.

# Chapter 2

# Parameterized Boolean Equation System

Before we discuss algorithms for simplifying PBESs, we first present the syntax and semantics of a PBES. The syntax and semantics follow those of [DPWar, GW05]. The algorithms we will discuss only use the syntax of PBESs. Although not used in our algorithms, the semantics are presented to give a better understanding of PBESs.

## 2.1 Syntax

In this section we describe the syntax of a PBES. A PBES is an equation system consisting of several parameterized boolean equations.

**Definition 2.1.1.** A parameterized boolean equation is defined as:

$\sigma X_i(\overrightarrow{d_i} : \overrightarrow{D_i}) = \varphi_i$

Here $\sigma$ is a fixed point operator which is either a $\nu$ (a greatest fixed point) or a $\mu$ (a least fixed point). $X_i \in \mathcal{X}$, where $\mathcal{X} = \{X_j | j \in \mathbb{N}\}$, is a variable; the solution of such a variable is a predicate, which is why $X_i$ is called a *predicate variable*. $\overrightarrow{d_i}$ is a vector of parameters, for which parameter $(\overrightarrow{d_i})_j$ (the $j^{th}$ parameter) has data sort $(\overrightarrow{D_i})_j$ (the $j^{th}$ data sort) . Every parameter name of an equation has to be unique for that equation. Finally, $\varphi_i$ is a predicate formula, which will be defined later.
During this thesis we will use *notes* to introduce conventions which we will use throughout the rest of the thesis. Here we present some conventions with regard to parameterized boolean equations:

*Note.*

- Whenever the fixed point sign of an equation is irrelevant, we use $\sigma$ as a fixed point operator.
- Whenever the term $\varphi_i$ or $\overrightarrow{d_i}$ is used in the context of an equation, the $i$ denotes the equation in which they occur; so $\varphi_i$ is the predicate formula that belongs to the equation for $X_i$ and

$\overrightarrow{d_i}$ is the vector of the parameters that belongs to the equation for $X_i$.

- In this thesis we consider vector $\overrightarrow{d_i}$ as a *set* when the order of the parameters does not matter (note that the parameters $\overrightarrow{d_i}$ are unique).

- If the number of parameters are not important we will use only one parameter; instead of parameter list $(\overrightarrow{d} : \overrightarrow{D})$, a single parameter $(d : D)$ is used.

We must still give the syntax of predicate formula $\varphi$, which we will do next.

**Definition 2.1.2.** The syntax of a predicate formula $\varphi$ is defined as:

$$\varphi ::= b \mid X(\overrightarrow{e}) \mid \varphi \oplus \varphi \mid Q_{d:D}.\varphi$$

Here $b$ is a data term of sort $\mathbb{B}$; an example of a data term is: $(n > 5 \wedge n \neq 10)$, where $n$ is a variable of type $\mathbb{N}$. $X$ is a predicate variable with as instantiation the terms $\overrightarrow{e}$. The operator $\oplus$ is either a conjunction ($\wedge$) or a disjunction ($\vee$). $Q$ is a quantification, either a universal quantification ($\forall$) or an existential quantification ($\exists$), where variable $d$ is a *fresh* variable of sort $D$. Note that negation does not occur in the syntax of a predicate formula, but it may be present as part of a data term. For convenience we allow implications ($\rightarrow$) after a data term $b$. This is valid because $b \rightarrow \varphi \equiv (\neg b) \vee \varphi$, and negation may be part of $b$. For an equation $\sigma X(\overrightarrow{d} : \overrightarrow{D}) = \varphi$ it must hold that all variables in $\varphi$ are bound in the scope of the equation $X$, either by a parameter or by a quantification. Outside the context of an equation it can happen that variables are not bound. To illustrate this we present a small example:
For equation $\sigma X(n : \mathbb{N}) = \forall_{m:\mathbb{N}}.n > m$, the predicate formula $\forall_{m:\mathbb{N}}.n > m$, contains variable $n$ which is not bound in this context. Furthermore, the predicate formula can be written as $\forall_{m:\mathbb{N}}.\varphi'$, where $\varphi' = n > m$. Here, in the context of $\varphi'$, there are two unbound variables, $m$ and $n$. For some definitions we will need the set of free (i.e. unbound) variables of a predicate formula $\varphi$.

*Note.* As a convention we will write $FV(\varphi)$ as the set of all free variables of predicate formula $\varphi$, in the context of $\varphi$.

Now that the syntax of a parameterized boolean equation is fully defined we can give a definition of a PBES

**Definition 2.1.3.** A parameterized Boolean Equation System $\mathcal{E}$ is defined as:

$$\mathcal{E} ::= \epsilon \mid (\sigma X_i(\overrightarrow{d_i} : \overrightarrow{D_i}) = \varphi_i)\mathcal{E}$$

A PBES is either $\epsilon$, which is an empty equation system, or an equation followed by an other PBES $\mathcal{E}$.

We want the constraint that predicate variables that occur in $\varphi$ are bound. Therefore we must first state what it means for a predicate variable to be bound.

**Definition 2.1.4.** For PBES $\mathcal{E}$, a predicate variable $X_i$ is bound if it occurs on the left hand side of an equation of $\mathcal{E}$.

If all predicate variables occurring in a predicate formula of an equation of PBES $\mathcal{E}$ are bound,

$\mathcal{E}$ is called *closed*; if there is an unbound predicate variable in $\mathcal{E}$, $\mathcal{E}$ is called *open*. In this thesis we only consider *closed* PBESs.

**Example 2.1.5.** We finish this section by presenting a *closed* PBES $\mathcal{E}$. This PBES will be used in some examples in later chapters.

$$\begin{array}{ll} \mu X_1(n_1, m_1 : \mathbb{N}) & = ((n_1 > 2) \to X_2(m_1 + n_1 + 1)) \wedge ((n_1 < 5) \to X_3(n_1 + 3)) \\ \nu X_2(n_2 : \mathbb{N}) & = \forall k : \mathbb{N}.(k > 3 \vee X_1(k, n_2)) \\ \nu X_3(n_3 : \mathbb{N}) & = X_1(n_3, n_3 + 1) \end{array}$$

This PBES is closed because all predicate variables that occur in a predicate formula (i.e. $\{X_1, X_2, X_3\}$) also occur in the left hand side of an equation.

## 2.2 Semantics

Here we present the semantics of PBESs. We start by describing the semantics of a predicate formula $\varphi$. This is given in the context of a certain data environment $\varepsilon$ and a certain predicate environment $\eta$. The data environment $\varepsilon$ assigns values to data variables. The predicate environment has type $\eta : \mathcal{X} \to (D \to \mathbb{B})$. $\eta$ takes a predicate $X_i \in \mathcal{X}$ as argument and returns a function with type $(D \to \mathbb{B})$. For an environment $\theta$, where $\theta$ is either $\eta$ or $\varepsilon$ we write $\theta[v/d]$ if we assign the value $v$ to the variable $d$. For example, if we want to assign 3 to the data variable $n$ in $\varepsilon$ we write $\varepsilon[3/n]$.

**Definition 2.2.1.** The semantics of a predicate formula is defined as:

$$\begin{array}{lllll} (1) & [\![b]\!]\ \eta\varepsilon & = & [\![b]\!]\varepsilon \\ (2) & [\![X(e)]\!]\ \eta\varepsilon & = & (\eta(X))([\![e]\!]\varepsilon) \\ (3) & [\![\varphi_1 \oplus \varphi_2]\!]\ \eta\varepsilon & = & [\![\varphi_1]\!]\eta\varepsilon \oplus [\![\varphi_2]\!]\eta\varepsilon \\ (4) & [\![Q_{d:D}.\varphi]\!]\ \eta\varepsilon & = & Q_{v:D}.[\![\varphi]\!]\eta(\varepsilon[v/d]) \end{array}$$

As no predicate variables occur in data terms, rule (1) states that the predicate environment is unimportant regarding the interpretation of a data term. The interpretation of a predicate variable $X$ with the term $e$ is the function $\eta(X) : D \to \mathbb{B}$ with the interpretation of $e$ applied on this function yielding either $true$ or $false$. The interpretation of the conjunction of two predicate formulas is the conjunction of the interpretation of the two predicate formulas. This is similar for disjunction and quantification.

**Example 2.2.2.** To illustrate how semantics of a predicate predicate formula can be used we present an example. Consider the data environment $\varepsilon$ and predicate environment $\eta$ which have the following assignments:

$$\begin{array}{l} \varepsilon(n) = 3 \\ \varepsilon(m) = 5 \\ \eta(X) = \lambda_{v,w:\mathbb{N}}.v < w \end{array}$$

We can now find the interpretation of $X(n+1, m+2)$ in the given environments:

$$
\begin{array}{ll}
 & [\![X(n+1,m+2)]\!]\eta\varepsilon \\
= & \quad \{(2)\} \\
 & (\eta(X))([\![(n+1,m+2)]\!]\varepsilon) \\
= & \quad \{\text{definition of } \eta\} \\
 & (\lambda_{v,w:\mathbb{N}}.v < w)([\![(n+1,m+2)]\!]\varepsilon) \\
= & \quad \{\text{definition of } \varepsilon, \text{ algebra}\} \\
 & (\lambda_{v,w:\mathbb{N}}.v < w)(4,7) \\
= & \quad \{\lambda\text{-calculus}\} \\
 & 4 < 7 \\
= & \quad \{\text{algebra}\} \\
 & \mathit{true}
\end{array}
$$

To be able to use parameters with the semantics of a predicate formula, we lift such a predicate formula to a function of type $D \to \mathbb{B}$. In the general case this function is:

$$\lambda_{v:D}.[\![\varphi]\!]\eta\varepsilon[v/d].$$

Here $d$ is a parameter which can be assigned a value using function application. This function will be used to give a definition of the semantics of a parameterized boolean equation.

The solution of a parameterized boolean equation is a function of type $D \to \mathbb{B}$. When the fixed point of a parameterized boolean equation is left out, the equation can have many solutions. For example, the equation $X(n:\mathbb{N}) = X(n)$ has infinitely many solutions (e.g. $\lambda_{v:\mathbb{N}}.\mathit{true}$, $\lambda_{v:\mathbb{N}}.v > 1$, $\lambda_{v:\mathbb{N}}.v > 2, \ldots$). With fixed point, the solution of an equation is a least ($\mu$) function or a greatest ($\nu$) function. For this we need an ordering on functions, which we will define next.

**Definition 2.2.3.** We denote the set of all functions of type $D \to \mathbb{B}$ as $\mathbb{B}^D$. Here we introduce a partial ordering on functions from $\mathbb{B}^D$:

$$f \sqsubseteq g \quad \Leftrightarrow \quad \forall_{d:D}.(f(d) \to g(d))$$

$f$ is *smaller* than $g$ iff $f \sqsubseteq g$, $f$ is *greater* than $g$ iff $g \sqsubseteq f$

$(\mathbb{B}^D, \sqsubseteq)$ is a complete lattice, this is proved in [GW05]. This means there is always a unique greatest fixed point and a unique least fixed point as the solution of an equation. Using the ordering $\sqsubseteq$ we can define the solution of a parameterized boolean equation.

**Definition 2.2.4.** For equation $\sigma X(d:D) = \varphi$ we define an operator $F$, which takes as input a function $f$ and returns the semantics of $\varphi$ lifted to a function *and* instantiates predicate variables $X$ in $\varphi$ with function $f$:

$$F = \lambda f : \mathbb{B}^D.\lambda_{v:D}.[\![\varphi]\!]\eta[f/X]\varepsilon[v/d]$$

A solution of a parameterized boolean equation without fixed point then is a function $g$ where:

$$g = (F)(g)$$

If $\sigma$ is a greatest fixed point $\nu$, the solution of the equation is the greatest function $g$; if $\sigma$ is a least fixed point $\mu$, the solution of the equation is the least function $g$. Here *greatest* and *least* are determined by the ordering $\sqsubseteq$.
As a shorthand notation we write $\nu g \in \mathbb{B}^D.F(g)$ as the greatest function $g$ for which $g = F(g)$ holds, and we write $\mu g \in \mathbb{B}^D.F(g)$ as the least function $g$ for which $g = F(g)$ holds.

We will now give an example of how Definition 2.2.4 can be used.

**Example 2.2.5.** Consider the equation $\mu X(n : \mathbb{N}) = n > 2 \wedge X(n)$. The function $F$ is defined as:

$$\begin{array}{ll} & F = \mu f \in \mathbb{B}^D.\lambda_{v:\mathbb{N}}.[\![n > 2 \wedge X(n)]\!]\eta[f/X]\varepsilon[v/n] \\ = & \quad \{\text{Environments}\} \\ & F = \mu f \in \mathbb{B}^D.\lambda_{v:\mathbb{N}}.v > 2 \wedge f(v) \end{array}$$

We now consider two solutions of this equation; note that here we are not taking the fixed point $\mu$ into account. A solution $g$ must have the following property:

$$g = (F)(g)$$

One solution for $g$ is $\lambda_{v:\mathbb{N}}.false$:

$$\begin{array}{ll} & \lambda_{v:\mathbb{N}}.false = (F)(\lambda_{v:\mathbb{N}}.false) \\ = & \quad \{\text{definition of } F\} \\ & \lambda_{v:\mathbb{N}}.false = (\lambda f \in \mathbb{B}^D.\lambda_{v:\mathbb{N}}.v > 2 \wedge f(v))(\lambda_{v:\mathbb{N}}.false) \\ = & \quad \{\lambda\text{-calculus}\} \\ & \lambda_{v:\mathbb{N}}.false = \lambda_{v:\mathbb{N}}.v > 2 \wedge (\lambda_{v:\mathbb{N}}.false)(v) \\ = & \quad \{\lambda\text{-calculus}\} \\ & \lambda_{v:\mathbb{N}}.false = \lambda_{v:\mathbb{N}}.v > 2 \wedge false) \\ = & \quad \{\text{logic}\} \\ & \lambda_{v:\mathbb{N}}.false = \lambda_{v:\mathbb{N}}.false) \\ = & \quad \{\text{logic}\} \\ & true \end{array}$$

An other solution for $g$ is $\lambda_{v:\mathbb{N}}.v > 2$:

$$
\begin{array}{ll}
 & \lambda_{v:\mathbb{N}}.v > 2 = (F)(\lambda_{v:\mathbb{N}}.v > 2) \\
= & \quad \{\text{definition of } F\} \\
 & \lambda_{v:\mathbb{N}}.v > 2 = (\lambda f \in \mathbb{B}^D.\lambda_{v:\mathbb{N}}.v > 2 \wedge f(v))(\lambda_{v:\mathbb{N}}.v > 2) \\
= & \quad \{\lambda\text{-calculus}\} \\
 & g = \lambda_{v:\mathbb{N}}.v > 2 \wedge (\lambda_{v:\mathbb{N}}.v > 2)(v) \\
= & \quad \{\lambda\text{-calculus}\} \\
 & \lambda_{v:\mathbb{N}}.v > 2 = \lambda_{v:\mathbb{N}}.v > 2 \wedge v > 2) \\
= & \quad \{\text{logic}\} \\
 & \lambda_{v:\mathbb{N}}.v > 2 = \lambda_{v:\mathbb{N}}.v > 2) \\
= & \quad \{\text{logic}\} \\
 & true
\end{array}
$$

This means there are two solutions for the formula $g = (F)(g)$, but using the relation $\sqsubseteq$ there is only one least ($\mu$) solution, which is $\lambda_{v:\mathbb{N}}.false$.

Now that we know the solution of a parameterized boolean equation, we can define the solution of a PBES.

**Definition 2.2.6.** The solution of a PBES in the context of a data environment $\varepsilon$ and a predicate environment $\eta$ is defined as:

$$
\begin{array}{lllll}
(1) & [\![\epsilon]\!]\eta\varepsilon & = & \eta \\
(2) & [\![(\sigma X(d:D) = \varphi)\mathcal{E}]\!]\eta\epsilon & = & [\![\mathcal{E}]\!](\eta[\sigma\ f \in \mathbb{B}^D.\lambda_{v:D}.[\![\varphi]\!](([\![\mathcal{E}]\!]\eta[f/X]\varepsilon)\varepsilon[v/d]/X)\varepsilon
\end{array}
$$

The solution of an empty PBES is defined to be the predicate environment, which is the assignment of a function of type $(D \to \mathbb{B})$ to each equation. The solution of a PBES with equation $(\sigma X(d : D) = \varphi)$ in front of $\mathcal{E}$ is $[\![\mathcal{E}]\!](\eta[f/X])\varepsilon$, which is the solution of $\mathcal{E}$ in a predicate environment where $X$ has the solution $f$. $f$ is the solution of equation $X$, which is: $\sigma\ f \in \mathbb{B}^D.\lambda_{v:D}.[\![\varphi]\!](([\![\mathcal{E}]\!]\eta[f/X]\varepsilon)\varepsilon[v/d]$. If $\sigma$ is $\mu$, $f$ is the least function for which $f = \lambda_{v:D}.[\![\varphi]\!](([\![\mathcal{E}]\!]\eta[f/X]\varepsilon)\varepsilon[v/d]$ holds and if $\sigma$ is $\nu$ it is the greatest function $f$ for which $f = \lambda_{v:D}.[\![\varphi]\!](([\![\mathcal{E}]\!]\eta[f/X]\varepsilon)\varepsilon[v/d]$ holds. Note that this formula is similar to $F$ of Definition 2.2.4, except now the solution of the rest of the PBES ($\mathcal{E}$) is taken into account as well.
To get a better understanding of Definition 2.2.6 we present an example.

**Example 2.2.7.** Here we will illustrate how to find a solution of a PBES. Consider the PBES

$$
\begin{array}{l}
\mu X_1(n_1 : \mathbb{N}) = X_2(n_1) \\
\nu X_2(n_2 : \mathbb{N}) = X_1(n_2)
\end{array}
$$

According to our definition we have:

$$
\begin{array}{ll}
 & [\![(\mu X_1(n_1 : \mathbb{N}) = X_2(n_1))(\nu X_2(n_2 : \mathbb{N}) = X_1(n_2))]\!]\eta\varepsilon \\
= & \quad \{\text{semantics of a PBES}\} \\
 & [\![\nu X_2(n_2 : \mathbb{N}) = X_1(n_2)]\!] \\
 & \quad (\eta[\mu\ f \in \mathbb{B}^{\mathbb{N}}.\lambda_{v:\mathbb{N}}.[\![X_2(n_1)]\!]\ ([\![\nu X_2(n_2 : \mathbb{N}) = X_1(n_2)]\!]\eta[f/X_1]\varepsilon)\ \varepsilon[v/n_1]\ \ /X_1)\varepsilon
\end{array}
$$

We now investigate the part: $([\![\nu X_2(n_2 : \mathbb{N}) = X_1(n_2)]\!]\eta[f/X_1]\varepsilon)$

$$
\begin{array}{ll}
 & [\![\nu X_2(n_2 : \mathbb{N}) = X_1(n_2)]\!]\eta[f/X]\varepsilon \\
= & \quad \{\text{semantics of a PBES}\} \\
 & [\![\epsilon]\!](\eta[\ \ \nu\ g \in \mathbb{B}^{\mathbb{N}}.\lambda_{w:\mathbb{N}}.[\![X_1(n_2)]\!]([\![\epsilon]\!]\eta[f/X_1][g/X_2]\varepsilon)\varepsilon[w/n_2]\ \ /X_2)\varepsilon \\
= & \quad \{\text{semantics of } [\![\epsilon]\!]\} \\
 & \eta[\nu\ g \in \mathbb{B}^{\mathbb{N}}.\lambda_{w:\mathbb{N}}.[\![X_1(n_2)]\!](\eta[f/X_1][g/X_2])\varepsilon[w/n_2]\ \ /X_2] \\
= & \quad \{\text{environments}\} \\
 & \eta[\nu\ g \in \mathbb{B}^{\mathbb{N}}.\lambda_{w:\mathbb{N}}.f(w)\ \ /X_2]
\end{array}
$$

We are now looking for the greatest fixed point $\nu$ of function $g = \lambda_{w:\mathbb{N}}.f(w)$. There is only one solution for $g$, which is the function $f$ itself. The solution to $([\![\nu X_2(n : \mathbb{N}) = X_1(n)]\!]\eta[f/X_1]\varepsilon)$ is thus $\eta[f/X_1][f/X_2]$.

$$
\begin{array}{ll}
 & [\![\nu X_2(n : \mathbb{N}) = X_1(n)]\!](\eta[\mu\ f \in \mathbb{B}^{\mathbb{N}}.\lambda_{v:\mathbb{N}}.[\![X_2(n)]\!]\eta[f/X_1][f/X_2]\varepsilon[v/n]\ \ /X_1)\varepsilon \\
= & \quad \{\text{semantics of predicate formula}\} \\
 & [\![\nu X_2(n : \mathbb{N}) = X_1(n)]\!]\eta[\mu\ f \in \mathbb{B}^{\mathbb{N}}.\lambda_{v:\mathbb{N}}.f[\![n]\!]\varepsilon[v/n]/X_1]\varepsilon \\
= & \quad \{\text{environments}\} \\
 & [\![\nu X_2(n : \mathbb{N}) = X_1(n)]\!]\eta[\lambda_{v:\mathbb{N}}.f(v)/X_1]\varepsilon
\end{array}
$$

We are now looking for the least solution ($\mu$) of $f$ in the equation $f = \lambda_{v:\mathbb{N}}.f(v)$. This can be calculated with approximation, which is $f = \lambda_{v:\mathbb{N}}.false$. This is the solution for equation $X_1$. When we fill this in we obtain:

$$[\![\nu X_2(n : \mathbb{N}) = X_1(n)]\!](\eta[(\lambda_{v:\mathbb{N}}.false)/X_1)\varepsilon$$

The derivation for the solution of $X_2$ is similar and is left to the reader. The solution of the PBES is $\eta[\lambda_{n:\mathbb{N}}.false/X_1][\lambda_{n:\mathbb{N}}.false/X_2]$. Note that should we swap the two equations the result is $(\nu X_2(n : \mathbb{N}) = X_1(n))(\mu X_1(n : \mathbb{N}) = X_2(n))$. The solution of the swapped PBES is different. The derivation is similar until we calculate $f$. The least fixed point that we calculated was $\lambda_{n:\mathbb{N}}.false$, but now we want the greatest fixed point, which is $\lambda_{n:\mathbb{N}}.true$. The solution to the PBES with the equations swapped is $\eta[\lambda_{n:\mathbb{N}}.true/X_2][\lambda_{n:\mathbb{N}}.true/X_1]$. This illustrates that the order of the equations is important in a PBES.

# Chapter 3

# Removing superfluous parameters

In this chapter we introduce the algorithm $ParElm$ which removes superfluous parameters from PBESs. A superfluous parameter is a parameter that cannot influence the solution of a PBES. E.g. parameter $n$ of PBES $\mu X(n : \mathbb{N}) = \varphi$ is superfluous if:

$$\forall_{i,j:\mathbb{N}}.[\![X(i)]\!]\eta\varepsilon = [\![X(j)]\!]\eta\varepsilon$$

Removing superfluous parameters can be very useful when using instantiation to solve a PBES. Instantiation transforms a PBES to a Boolean Equation System (BES)[Mad97], which is a PBES without data parameters. The number of boolean equations in the BES created by instantiation is partly determined by the number and type of parameters of a PBES. For instance, removing one parameter of type $\mathbb{B}$ from a PBES and then using instantiation can result in halving the number of equations of the resulting BES, which greatly reduces the time needed to find a solution. When removing a parameter with a certain type that has more elements than a type $\mathbb{B}$, the reduction in the resulting BES can be even greater.
Instantiation is a technique that is undecidable in general. When used on a PBES with only finite data types however, it can always compute a PBES. If $ParElm$ is able to remove all infinite data types from a PBES, this PBES can be solved using instantiation. Because of this more PBESs can be solved when first applying $ParElm$. This makes $ParElm$ a useful technique when using instantiation.

A parameter is superfluous if it does not affect the solution of a PBES, like with the PBES $\nu X(n : \mathbb{N}) = false$, the parameter $n$ is superfluous. Note that the solution to $X$ is $\lambda v : \mathbb{N}.false$, which is strictly not the same as $false$. The PBES $\nu X = false$ does have $false$ as solution. We still wish to consider both PBESs to be equal since, apart from their typing, both solutions are indistinguisable, i.e.:

$$\forall_{n:\mathbb{N}}.((\lambda v : \mathbb{N}.false)(n) = false)$$

The goal of $ParElm$ is to compute a set of parameters that does not influence the solution of a PBES. This is achieved by determining which parameters *might* influence the solution of a PBES. Section 3.1 presents a technique to find these influential parameter. Note that there already is an algorithm for finding superfluous parameters in process descriptions [GL02] (LPS $ParElm$). This has inspired some aspects of $ParElm$ on a PBES level. However, some techniques introduced in Section 3.1 can be brought back to improve the $ParElm$ algorithm for processes. In Section

3.2, improved LPS $ParElm$ is compared to the current LPS $ParElm$ algorithm. Finally, some tests and results of an implementation of $ParELm$ for PBESs are presented in Section 3.3

## 3.1 Parameter reduction

The solution of a PBES is a function assignment to every predicate variable in the equation system. If the solution of an equation changes then so does the solution of the PBES. A parameter $d$ can change the solution of an equation if it occurs in a data term $b$, or it can influence the solution by changing the solution of a predicate variable $X(e)$. This is the case since a parameter can only be present in data terms and predicate variable instantiations. We will investigate both these cases in the next sections.

### 3.1.1 Influential parameters in a data term

Just by inspecting a data term $b$ it cannot be decided if parameter $d$ that occurs $b$ is influential. To illustrate this, consider the equation $\mu X(n : \mathbb{N}) = n > 2 \wedge X_2$, parameter $n$ occurs in conjunction with a predicate variable. If the solution of predicate variable $X_2$ is $false$, the solution of $X_1$ is the function $\lambda_{v:\mathbb{N}}.false$. Here it is clear that $n$ is not an influential parameter. To know this, the solution of $X_2$ is required, which has to be calculated. This is not desired because we want to run $ParElm$ before the calculation of a PBES. This is why we stay on a syntactic level when determining the set of influential parameters.
On a syntactic level we usually cannot tell if a parameter that occurs in a data term is influential. Therefore we use an over-approximation of the set of influential parameters. The approximated set is the set of *all* parameters that occur in data terms. Sometimes we *can* syntactically determine that a parameter in a data term is *not* influential. This is usually the case when the predicate formula can be simplified using some simple rules like:

- $\varphi \wedge false \equiv false$
- $\varphi \vee true \equiv true$
- $\varphi \wedge \neg\varphi \equiv false$
- $\varphi \vee \neg\varphi \equiv true$

Because of this, simplifying a PBES before using $ParElm$ can help to find more parameters that can be eliminated. Here we give a formal definition of all influential parameters that occur in data terms of a PBES. For this, we first introduce a definition of influential parameters in a single predicate formula. These are *all* parameters that occur in a data term of a predicate formula.

**Definition 3.1.1.** The parameters that occur in a data term $b$ of predicate formula $\varphi$ is $DP_f(\varphi)$, where $DP_f(\varphi)$ is defined as:

$$\begin{aligned} DP_f(b) &= FV(b) \\ DP_f(X(\overrightarrow{e})) &= \emptyset \\ DP_f(\varphi_1 \oplus \varphi_2) &= DP_f(\varphi_1) \cup DP_f(\varphi_2) \\ DP_f(Q_{d:D}.\varphi) &= DP_f(\varphi) \backslash \{d\} \end{aligned}$$

Note that variables bound by a quantification are excluded from the set. This is because we are only interested in *parameters*. Since all variables in an equation are bound and we omit the variables bound by a quantification, $DP_f(\varphi)$ is indeed the set of all *parameters* which occur in data terms of $\varphi$.

**Example 3.1.2.** Definition 3.1.1 used on the PBES of Example 2.1.5 results in:

$DP_f(\varphi_1) = \{n_1\}$ because $n_1$ occurs in a data term and $m_1$ does not occur in a data term.
$DP_f(\varphi_2) = \emptyset$ because $n_2$ does not occur in a data term.
$DP_f(\varphi_3) = \emptyset$ because $n_2$ does not occur in a data term.

The set of influential parameters of a PBES that occur in data terms, is simply the union of all influential parameters that occur in all predicate formulas of the PBES. Here we formally define this set.

**Definition 3.1.3.** For PBES $\mathcal{E}$, the set $DP_p(\mathcal{E})$ of influential parameters in data terms of PBES $\mathcal{E}$ is defined as:

$$DP_p(\epsilon) = \emptyset$$
$$DP_p((\sigma X(d:D) = \varphi)\mathcal{E}) = DP_f(\varphi) \cup DP_p(\mathcal{E})$$

**Example 3.1.4.** The set $DP_p(\mathcal{E})$ of the PBES of Example 2.1.5 is:

$$\{n_1\} \cup \emptyset \cup \emptyset = \{n_1\}$$

With $DP_p(\mathcal{E})$ a set can be constructed containing all influential parameters in data terms of PBES $\mathcal{E}$.

### 3.1.2 All influential parameters

It can also be the case that a parameter $d$ influences the solution of a PBES when it occurs in a term $e$ used in a predicate variable instantiation $X(e)$. For example, in the PBES $(\nu X_1(n_1 : \mathbb{N}) = X_2(n_1 + 3))(\mu X_2(n_2 : \mathbb{N}) = n_2 > 2)$, $n_1$ occurs in the term $n_1 + 3$ of predicate variable instantiation $X_2(n_1+3)$. In this situation $n_2$ is instantiated with $n_1+3$ , so $n_1$ influences $n_2$. This does not mean $n_1$ influences the solution of the PBES, $n_1$ only influences the solution if $n_2$ itself is influential. Here the main problem arises; to determine if a parameter is influential, we might need to check if one or more other parameters are influential. This means there are (possibly circular) dependencies between parameters.
To solve this problem we compute which parameter influences which other parameter and combine these relations in an *influential graph*. Such a graph has parameters as vertices and directed

edges $(d_i, d_j)$ which represent that a parameter $d_i$ influences parameter $d_j$.
When parameter $d_1$ influences parameter $d_2$, $d_1$ is influential if $d_2$ is influential. Parameter $d_2$ is influential when it occurs in a data term (see Section 3.1.1), or if $d_2$ influences an influential parameter $d_3$. This can be computed using the influential graph; If a parameter $d_i$ can influence (reach in the graph) an influential parameter $d_j$, then $d_i$ is influential too. Since we know which parameters are influential using Section 3.1.1, we can calculate all influential parameters that occur in predicate variables. When we combine these parameters with the ones already calculated in Section 3.1.1, we have the set of all influential parameters.

To help construct the influential graph we first introduce a definition that defines a set of all predicate variables instantiations that occur in a predicate formula.

**Definition 3.1.5.** The set $PV(\varphi)$ contains all predicate variable instantiations of $\varphi$ and is defined as:

$$\begin{aligned} PV(b) &= \emptyset \\ PV(X(\overrightarrow{e})) &= \{X(\overrightarrow{e})\} \\ PV(\varphi_1 \oplus \varphi_2) &= PV(\varphi_1) \cup PV(\varphi_2) \\ PV(Q_{d:D}.\varphi) &= PV(\varphi) \end{aligned}$$

**Example 3.1.6.** Using Definition 3.1.5 on the equations of Example 2.1.5 results in:

$$\begin{aligned} PV(\varphi_1) &= \{X_2(m_1 + n_1 + 1), X_3(n_1 + 3)\} \\ PV(\varphi_2) &= \{X_1(k, n_2)\} \\ PV(\varphi_3) &= \{X_1(n_3, n_3 + 1)\} \end{aligned}$$

Next we define the influential graph that is associated to a PBES.

**Definition 3.1.7.** The influential graph for PBES $(\sigma X_1(\overrightarrow{d_1} : \overrightarrow{D_1} = \varphi_i) \dots (\sigma X_n(\overrightarrow{d_n} : \overrightarrow{D_n}) = \varphi_n)$ is the directed graph $G = (V, E)$ where:

$$V = \bigcup_i \overrightarrow{d_i}$$

$$E = \{(\overrightarrow{d_i})_a, (\overrightarrow{d_j})_b)) | X_j(\overrightarrow{e}) \in PV(\varphi_i) \wedge (\overrightarrow{d_i})_a \in FV((\overrightarrow{e})_b)\}$$
where $i, j \in \{1, \dots, n\}, a \in \{1, \dots, \#(\overrightarrow{d_i})\}$, and $b \in \{1, \dots, \#(\overrightarrow{e})\}$

This graph captures which parameters influence which other parameters in a single step. We can use this information to compute set of influential parameters in predicate variables. First we present an example of Definition 3.1.7.

**Example 3.1.8.** Using Definition 3.1.7 to construct an influential graph of the PBES of Example 2.1.5 results in:

$$\begin{aligned} V &= \{n_1, m_1, n_2, n_3\} \\ E &= \{(m_1, n_2), (n_1, n_3), (n_1, n_2), (n_2, m_1), (n_3, n_1), (n_3, m_1)\} \end{aligned}$$

$V$ is the set of all parameters that occur in the PBES of Example 2.1.5 and $E$ is the set of directed edges. Each edge $(d_i, d_j)$ states that parameter $d_i$ influences parameter $d_j$. The visual representation of this graph is presented in Figure 3.1.

Using an influential graph of a PBES $\mathcal{E}$, together with the set of influential parameters $DP_p(\mathcal{E})$,

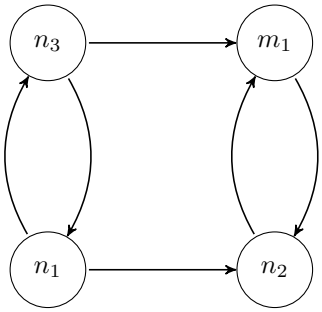



Figure 3.1: Influential graph of Example 2.1.5

we can calculate the set of influential predicate variables. This is a reachability problem, which can for instance be solved by taking the transitive closure of the influential graph and marking all parameters with an edge to a parameter $d_i \in DP_p(\mathcal{E})$ as influential. To get the set of all influential parameters that occur in either a predicate variable or a data term we can combine these two sets. Another way is by taking the transitive, *reflexive* closure of the influential graph and marking all parameters that have an edge to parameter $d_i \in DP_p(\mathcal{E})$ as influential.

**Definition 3.1.9.** Let $G = (V, E)$ be an influential graph of PBES $\mathcal{E}$. Let $E'$ be the transitive, reflexive closure of $E$. The set $\mathcal{I}(\mathcal{E})$ of influential parameters is defined as:

$$\mathcal{I}(\mathcal{E}) = \{d_i | (d_i, d_j) \in E' \wedge d_j \in DP_p(\mathcal{E})\}$$

The set $\mathcal{I}(\mathcal{E})$ is the set of parameter which directly or indirectly might influence the solution of PBES $\mathcal{E}$.

*Note.* Finding influential parameters is a reachability problem. Definition 3.1.9 uses the transitive reflexive closure to define the set of influential parameters. This method has time complexity $\mathcal{O}(n^3)$, where $n$ are the number of vertices. Such an algorithm is presented in [CLRS01]. In the implementation of Appendix A a more efficient technique is used, which has time complexity $\mathcal{O}(e)$, where $e$ is the number of edges.

When a parameter is in the set $\mathcal{I}(\mathcal{E})$, it is *not* guaranteed that it is influential; this is due to the fact that we are using an over-approximated set $DP_p(\mathcal{E})$. All parameters that are not in the set $\mathcal{I}(\mathcal{E})$ are superfluous. The set $\mathcal{S}(\mathcal{E})$ is the dual of the set $\mathcal{I}(\mathcal{E})$, that is, for all parameters $d$ in PBES $\mathcal{E}$, $d \in \mathcal{I}(\mathcal{E}) \Leftrightarrow d \notin \mathcal{S}(\mathcal{E})$.

**Example 3.1.10.** To illustrate how to find all the parameters of $\mathcal{S}(\mathcal{E})$ we will use the PBES $\mathcal{E}$ of Example 2.1.5. Figure 3.2a presents the influential graph of $\mathcal{E}$, where the parameters from $DP_p(\mathcal{E})$ are marked. In this case only one parameter is marked as influential, but all parameters that can reach an influential parameter are indirectly influential too. Using Definition 3.1.9 we can do a reachability test. In this case $n_3$ can affect the value of influential parameter $n_1$, so it is marked as influential. Figure 3.2b presents the set $\mathcal{I}(\mathcal{E})$. The set $\mathcal{S}(\mathcal{E})$ consists of all parameters that are not marked. All parameters of $\mathcal{S}(\mathcal{E}) = \{m_1, n_2\}$ are superfluous.

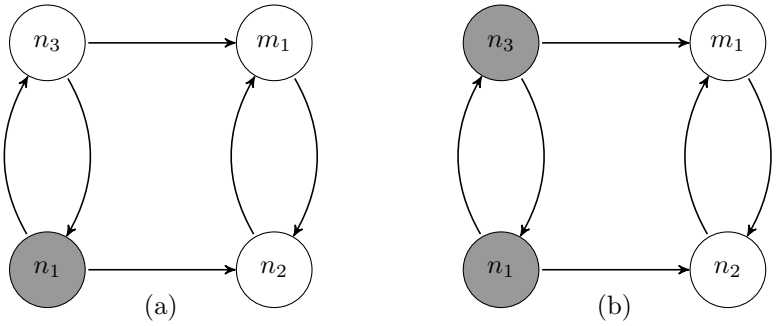



Figure 3.2: Influential graph of Example 2.1.5: (a) parameters from the set $DP_p(\mathcal{E})$ marked, (b) parameters of $\mathcal{I}(\mathcal{E})$ marked

### 3.1.3 Removing parameters

The previous sections described how we can find the set $\mathcal{S}(\mathcal{E})$ of superfluous parameters. All superfluous parameters can be removed from a PBES without affecting the solution (apart from the typing). Here we give a definition of how this is done.

**Definition 3.1.11.** Assume $S$ is a set of superfluous parameters. Furthermore $\forall_i(\overrightarrow{d})_i \in S$ and $\forall_i(\overrightarrow{e})_i \notin S$. Without loss of generality, we assume that in the definition of an equation, the superfluous parameters $\overrightarrow{d}$ are in front of the influential parameters $\overrightarrow{e}$. This is valid since we known the set of superfluous/influential parameters and, if parameters of all predicate variables $X_i$ are swapped in the same way, the solution of a PBES does not change. Now we can define $PE(\mathcal{E})$, where $\mathcal{E}$ is a PBES:

$$
\begin{array}{ll}
PE(\epsilon) & = \varepsilon \\
PE((\sigma X(\overrightarrow{d} : \overrightarrow{D}, \overrightarrow{e} : \overrightarrow{D}) = \varphi)\mathcal{E}) & = (\sigma X(\overrightarrow{e} : \overrightarrow{D}) = Rem(\varphi, S))PE((\mathcal{E}) \\
Rem(b) & = b \\
\quad \text{For } \#(\overrightarrow{a}) = \#(\overrightarrow{d_i}) \text{ and } \#(\overrightarrow{b}) = \#(\overrightarrow{e_i}) & \\
Rem(X_i(\overrightarrow{a}, \overrightarrow{b})) & = X(\overrightarrow{b}) \\
Rem(\varphi_1 \oplus \varphi_2) & = Rem(\varphi_1) \oplus Rem(\varphi_2) \\
Rem(Q_{d:D}.\varphi) & = Q_{d:D}.Rem(\varphi)
\end{array}
$$

In the next example we demonstrate the methods described in this chapter by applying them on a larger example.

**Example 3.1.12.** Here we demonstrate all *ParElm* steps needed to reduce a PBES. The PBES we are using is:

$$
\begin{array}{ll}
\mu X_1(n_1, m_1, l_1 : \mathbb{N}) & = n_1 > 2 \wedge (X_2(m_1) \vee X_3(l_1 > 2)) \\
\mu X_2(n_2 : \mathbb{N}) & = \forall m : \mathbb{N}.X_5(n_2, n_2 + m) \\
\nu X_3(b_3 : Bool) & = b_3 \wedge \exists m : \mathbb{N}.X_4(m, m + 1) \\
\mu X_4(n_4, m_4 : \mathbb{N}) & = X_5(n_4, m_4) \vee X_1(n_4, 5, 5) \\
\nu X_5(n_5, m_5 : \mathbb{N}) & = X_4(n_5, m_5)
\end{array}
$$

Using Definition 3.1.3 we can determine the set $DP_p(\mathcal{E})$:

$$DP_f(\varphi_1) \cup DP_f(\varphi_2) \cup DP_f(\varphi_3) \cup DP_f(\varphi_4) \cup DP_f(\varphi_5) = \{n_1\} \cup \emptyset \cup \{b_3\} \cup \emptyset \cup \emptyset = \{n_1, b_3\}.$$

The parameters $DP_p(\mathcal{E})$ directly influence the solution of $\mathcal{E}$; to determine which parameters are indirectly influential we construct the associated influential graph. Using Definition 3.1.9 we obtain the graph $G = (V, E)$ where:

$V = \{n_1, m_1, l_1, n_2, b_3, n_4, m_4, n_5, m_5\}$
$E = \{(m_1, n_2), (l_1, b_3), (n_2, n_5), (n_2, m_5), (n_4, n_5), (m_4, m_5), (n_4, n_1), (n_5, n_4), (m_5, m_4)\}$

The result obtained by marking the parameters of $DP_p(\mathcal{E})$ in graph $G$ is presented in Figure 3.3. The set $\mathcal{I}(\mathcal{E})$ is made by marking all states that can reach an influential parameter as influential. The result is presented in Figure 3.4. From this graph we can conclude that the parameters $\{m_4, m_5\} \in \mathcal{S}(\mathcal{E})$ may be removed. This can be done using Definition 3.1.11; The resulting PBES is:

$$\begin{array}{ll}
\mu X_1(n_1, m_1, l_1 : \mathbb{N}) & = n_1 > 2 \wedge (X_2(m_1) \vee X_3(l_1 > 2)) \\
\mu X_2(n_2 : \mathbb{N}) & = \forall m : \mathbb{N}. X_5(n_2) \\
\nu X_3(b_3 : \mathit{Bool}) & = b_3 \wedge \exists m : \mathbb{N}. X_4(m) \\
\mu X_4(n_4 : \mathbb{N}) & = X_5(n_4) \vee X_1(n_4, 5, 5) \\
\nu X_5(n_5 : \mathbb{N}) & = X_4(n_5)
\end{array}$$

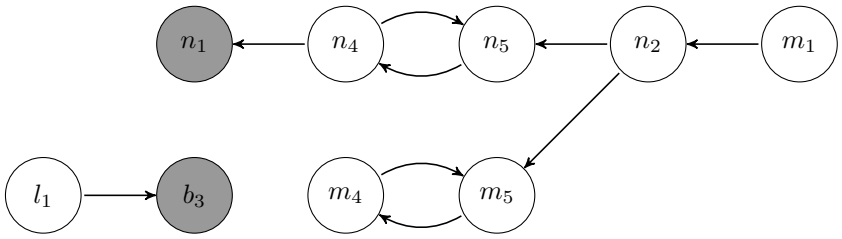



Figure 3.3: Influential graph of Example 3.1.12, with parameters from the set $DP_p(\mathcal{E})$ marked

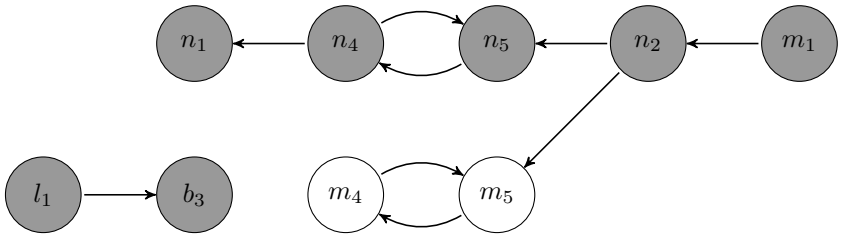



Figure 3.4: Influential graph of Example 3.1.12, with all influential parameters marked

## 3.2 ParElm for LPSs

An algorithm for removing superfluous parameters from an LPS is proposed in [GL02], and implemented in the mCRL2 tool set [GMR+07]. This version shares the same idea as *ParElm* for PBESs; it marks the set of parameters that directly influences the solution of an LPS as influential, and searches for parameters that indirectly influences the solution of an LPS by checking if a parameter influences a marked parameter. In the mCRL2 version, however, finding the set of parameters that indirectly influence the solution of an LPS is not done with an influential graph. Currently this is done by recursively checking all parameters and marking every parameter that instantiates an influential parameter. This results in quadratic time complexity (quadratic in the number of parameters). Since a reachability test in graphs takes linear time (linear in the number of edges), the LPS *ParElm* can profit from the influential graph technique. An implementation of LPS *ParElm* using a influential graph has been created; a comparison with the mCRL2 version is presented in Section 3.2.2. We first present the syntax of LPSs and describe how to construct an LPS *ParElm* algorithm in the next section.

### 3.2.1 Linear process specifications

An LPS describes a process using process algebra. The advantage of an LPS is that it is always in a certain (simple) form. Some advantages of this form is that it contains no parallelism and there is only one recursion variable. Here we present the syntax of an untimed linear process:
$X(d : D) = \sum_{i \in I} \sum_{e:E_i} c_i(d, e) \rightarrow a_i(f_i(d, e))X(g_i(d, e)) + \sum_{i \in J} \sum_{e:E_i} c_i(d, e) \rightarrow \delta$

We will not go into detail about the syntax and semantics of LPSs. More information about LPSs can for instance be found in [GR01]. The parts that are important for *ParElm* are:

- $c_i$: a condition that may contain parameters
- $a_i$: an action that may contain parameters
- $X$: a process parameter, if used on the right hand side it instantiates the process $X$ with values which may depend on parameters.

From the definition of LPSs we can conclude that a parameter can occur in an action, a condition or an instantiation of a linear process. The set of parameters that are directly influential are all parameters that occur in either a condition or an action. The set of parameters that indirectly influence the solution of an LPS can be calculated with an influential graph. The construction of this graph is similar as it is for PBESs; the set of vertices is the set of all parameters, and the directed edges can be extracted from the process parameter instantiations. Using the influential graph all influential parameters can be calculated. All parameters that are not influential can be removed from the LPS.

### 3.2.2 Comparison

In this section we will compare *ParElm* for LPSs which is currently distributed with the mCRL2 tool set (up to revision 4833) with a new implementation of *ParElm* for LPS using an influential graph. First we note that both versions find the same superfluous parameters. This is because the general idea of which parameters are influential is the same; the only difference is the time complexity of the implemented algorithms. For the comparison we will construct LPSs, each

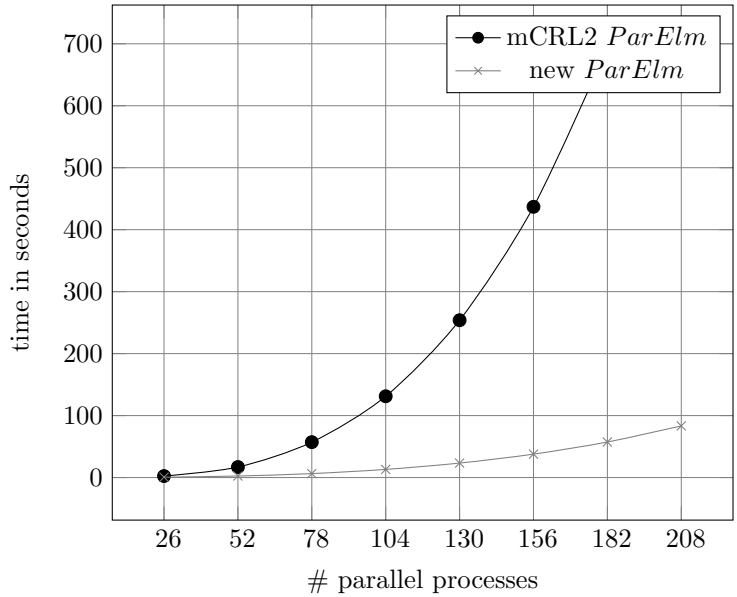



Figure 3.5: Running time comparison of the implementation of the mCRL2 tool set and LPS *ParElm* using an influential graph

with increased numbers of parallel processes. The test case we present here has the following mCRL2 specification:

```
act
  a1,a2: Nat;

proc
  P(d1,d2:Nat) = a1(d1).a2(d2).P(d1,d2);

 init
  P(0,0) || ... || P(n,n)
```

Here $init = P(0,0)||\ldots||P(n,n)$, which are $n$ processes of `P` in parallel. When $n$ increases, the number of parameters in the resulting LPS increases linearly. Using the mCRL2 specification described above we can construct LPSs for varying values of $n$, and apply both the LPS *ParElm* algorithms. The running time is measured; the results are presented in Figure 3.5. From this figure we can conclude that the algorithm using the influential graph performs better for big numbers of $n$, which is due to the linear running time of the algorithm.

## 3.3 Example using *ParElm*

In this section we show the importance of *ParElm* using the mCRL2 tool set [GMR+07]. As an example we will use the alternating bit protocol (ABP) which sends data from a sender, through a medium that nondeterministically garbles messages or not, to a receiver. Until now the mCRL2 toolkit could only verify properties of the ABP if there is a finite set of data which can be sent,

i.e. it can only send data from the set of data messages $D$, where $\{d_1, d_2, \ldots, d_n\} \in D$ for finite $n$. Using *ParElm* we can also prove some properties of the ABP where we consider arbitrary data, in this case from the infinite set $\mathbb{N}$.
As we are using the mCRL2 tool set, we start with a mCRL2 specification of the ABP. A version of the ABP is present as an example in the mCRL2 tool set. This version works, as stated above, with a finite set $D$. We change the definition of $D$ from:

```
D = struct d_1 | d_2;
```

to:

```
D = Nat;
```

The complete mCRL2 specification of the ABP is presented in Appendix C. We will now make four PBESs, each encoding a property we wish to check about the ABP specification. These four properties are specified with modal $\mu$-calculus formulas. These properties are the same as defined in [GW04], except that some errors have been fixed:

Property 1: No deadlock may occur

```
nu X. ([true]X && <true>true)
```

Property 2: A message that is sent can always eventually be received

```
nu X. ([true]X && forall d:Nat. [r1(d)] mu Y. (<true>Y || <s4(d)>true))
```

Property 3: The protocol does not generate new messages

```
forall d:Nat. nu X. (([!r1(d)]X && [s4(d)]false))
```

Property 4: The protocol does not duplicate messages

```
[true*] forall d:D. [r1(d).(!r1(d) && !s4(d))*.s4(d).(!r1(d))*.s4(d)]false
```

Using the mCRL2 tool set we can make a PBES from a mCRL2 specification and a modal $\mu$ calculus formula. Using the four formulas presented above we make four PBESs and apply *ParElm* on them. Creating the PBESs is done with the tools *mcrl22lps* and *lps2pbes*. Note that there is an intermediate LPS file, on which we can apply *ParElm* for LPSs as discussed in Section 3.2, but no parameters can be removed using this technique. Note that the resulting PBESs have more parameters than the mCRL2 specification. These parameters are introduced to make the intermediate LPS file. All tests are done on a Windows machine with a 1.8Ghz Pentium M processor and 1GB of RAM. The results are presented below:

| Property | Running time | Parameters removed | Parameters remaining | Type |
|---|---|---|---|---|
| 1 | 108 ms | 3 | 0 | Nat |
| | | 0 | 4 | Pos |
| | | 0 | 4 | Bool |
| 2 | 125 ms | 0 | 7 | Nat |
| | | 0 | 8 | Pos |
| | | 0 | 8 | Bool |
| 3 | 110 ms | 0 | 7 | Nat |
| | | 0 | 8 | Pos |
| | | 0 | 8 | Bool |
| 3 | 125 ms | 0 | 7 | Nat |
| | | 0 | 8 | Pos |
| | | 0 | 8 | Bool |

From this table we can see that the running time of *ParElm* is low, while for the PBES encoding the model checking problem for property 1, all parameters of type *Nat* are removed. The parameters with type Pos are added by the tool *mcrl22lps* and only use a finite subset of numbers from the set of positive integers. Because of this there are only parameters left with finite data for this PBES. The result is that this PBES can always be solved, for example by using instantiation [DPWar]. For the other three PBESs other techniques must be applied because it depends on infinite type *Nat*. Symbolic approximation [GW04] is a technique that can be used to solve these PBESs. *ParElm* is also tested on other PBESs and is able to remove parameters for a number of them. This makes it fruitful to run *ParElm* before trying to solve a PBES.

# Chapter 4

# Detecting constants

It is not always the case that we want to know the solution of a whole PBES. When a PBES is created from a process specification and a modal $\mu$-calculus formula, we usually only want to know if $X_1(\overrightarrow{e})$ holds. Here $X_1$ is the predicate variable of the first equation and $\overrightarrow{e}$ are data terms which encodes the initial state of the used process. If $X_1(\overrightarrow{e})$ is $true$, the $\mu$-calculus formula holds for the given process. $ConstElm$ can use the information of $X_1(\overrightarrow{e})$ to reduce a PBES in such a way that the solution of $X_1(\overrightarrow{e})$ does not change. Note that solutions of other predicate variables may change *if* this does not influence the solution of $X_1(\overrightarrow{e})$. A PBES is not only used to verify properties of processes, this is why we generalize the specification of $ConstElm$. Given a predicate $\kappa$, where $\kappa$ contains predicate variables of PBES $\mathcal{E}$, $ConstElm(\mathcal{E}, \kappa)$ will try to reduce $\mathcal{E}$ in such a way that the solution of $\kappa$ does not change:

$$[\![\kappa]\!]([\![\mathcal{E}]\!]\eta\varepsilon)\varepsilon \equiv [\![\kappa]\!]([\![ConstElm(\mathcal{E}, \kappa)]\!]\eta\varepsilon)\varepsilon$$

In the predicate $\kappa$, the parameters of some predicate variables are instantiated with concrete values. What $ConstElm$ does is determining if parameters instantiated by these values are constants. If this is the case, parameters in the PBES can be replaced with constants. When some parameters are constant and used in predicate variable instantiations, even more constants can be found. A parameter that is constant can be considerd to be an invariant of a PBES. As explained in [OW08], invariants are very useful when using symbolic approximation [GW04] to solve a PBES. How constants can be detected will be explained in Section 4.1. Some tests and results are presented in Section 4.2

## 4.1 The algorithm $ConstElm$

In this section we define the algorithm $ConstElm$. The goal of this algorithm is to find parameters $d$ for which the invariant $d = v$ holds, where $v$ is a constant value. Such an invariant holds if parameter $d$ will always be instantiated with the same constant value $v$. If this is the case the parameter $d$ can be replaced with constant $v$. Furthermore we only have to make sure that the solution of $\kappa$ does not change. Because of this we can replace a parameter by a constant if the parameter remains constant *during the calculation of* $\kappa$. To illustrate this we give an example:

**Example 4.1.1.** Take $\kappa = X_1(1)$ with PBES $\mathcal{E}$ =

$$\begin{array}{ll} \mu X_1(n_1 : \mathbb{N}) = & n_1 > 2 \\ \nu X_2 \qquad\quad = & X_1(4) \end{array}$$

Parameter $n_1$ is instantiated with the value 1, taken from predicate $\kappa$. Parameter $n_1$ is also instantiated with the value 4 by predicate variable instantiation $X_1(4)$ of the second equation. Since the second equation is not needed for the calculation of $X_1(1)$ it can be safely ignored. $ConstElm$ can reduce $\mathcal{E}$ to $\mathcal{E}$'=

$$\begin{array}{ll} \mu X_1 = & 1 > 2 \\ \nu X_2 = & X_1 \end{array}$$

Here $[\![\kappa]\!]\mathcal{E}\eta\varepsilon = [\![\kappa]\!]\mathcal{E}'\eta\varepsilon = false$. The solution of the PBES has changed, but the solution for $\kappa$ has not. We can even do one more reduction; since the second equation is not needed to calculate the solution of $\kappa$, it can be removed.

We cannot tell beforehand which equations are needed to calculate the solution of a predicate $\kappa$. Section 4.1.1 presents a method that will detect if parameters are constant when the solution of $\kappa$ may not change. Section 4.1.2 introduces a way to find more constants by inspecting data terms that can be evaluated using the known constants.

### 4.1.1 Finding constants

Because parameters are instantiated by predicate variables, we treat predicate variables as assignments; if a predicate variable $X$ instantiates parameter $d$ with as value the term $e$, we consider $X$ to be the assignment $d := e$. For instance with PBES $(\mu X_1(m_1, n_1 : \mathbb{N}) = X_2(n_1, 4))(\nu X_2(m_2, n_2 : \mathbb{N}) = n_2 > m_2)$, the predicate variable instantiation $X_2(n_1, 4)$ is treated as the assignment $m_2, n_2 := n_1, 4$, as the parameters of the equation for $X_2$ are instantiated with the values $n_1$ and 4 respectively. In this example, parameters of the equation for $X_2$ are assigned values from the equation for $X_1$. If values of the parameters of the equation for $X_1$ are known we can use the assignments to inspect the parameters of the equation for $X_2$. For this reason we make a graph with predicate variables as vertices and assignments as edges. If values of the parameters of the equation for $X_i$ are known we should check all the edges starting from vertex $X_i$. This way updates are done on a local level. Such a graph shows the dependency of one equation to another equation. Therefore we call this a dependency graph.

**Definition 4.1.2.** A dependency graph $G = (V, E, L)$ of a PBES $\mathcal{E}$, where $V$ are the vertices, $L$ is a set of labels and $E \in (V \times l \times V)$ is a set of labeled directed edges, is defined as:

$$\begin{array}{ll} V = \bigcup_{i=1}^{n} \{X_i\} & \\ L = \{\overrightarrow{d_j} := \overrightarrow{e} \mid X_j(\overrightarrow{e}) \in PV(\varphi_i)\} & \\ E = \{(X_i, l, X_j)\} \text{ where } l \in L\} & \text{if } X_j(\overrightarrow{e}) \in PV(\varphi_i) \end{array}$$

Here the vertices $V$ are the predicate variables, $L$ is the set of assignments and $E$ is a set of labeled directed edges, which represents the instantiations of parameters of equation $X_j$ using label $l$ from equation $X_i$.

*Note.* The label $l$ of edge $(X_i, l, X_j)$ is a list of assignments. For the remainder of this article we will use the word *assignment* instead of *label*.

**Example 4.1.3.** Here we give an example of the dependency graph of a given PBES $\mathcal{E} =$

$$
\begin{array}{ll}
\mu X_1(n_1, m_1, o_1, p_1 : \mathbb{N}) & = (n_1 \leq m_1 \vee X_2(o_1, p_1)) \wedge X_3(n_1) \wedge X_1(n_1, m_1, 4, p_1) \\
\mu X_2(n_2, m_2 : \mathbb{N}) & = X_5(n_2, m_2) \vee X_5(m_2, n_2) \\
\nu X_3(n_3 : \mathbb{N}) & = n_3 \leq 3 \vee X_1(n_3, n_3, 4, n_3 + 1) \\
\nu X_4(n_4, m_4, o_4 : \mathbb{N}) & = n_4 \leq (m_4 + o_4) \vee (X_3(n_4) \wedge X_4(n_4, m_4 + 1, n_4)) \\
\mu X_5(n_5, m_5 : \mathbb{N}) & = n_5 > m_5 \vee X_3(n_5)
\end{array}
$$

According to Definition 4.1.2, $G = (V, E, L)$, where

$V = \{X_1, X_2, X_3, X_4, X_5\}$
$L = \{$
$(n_2, m_2 := o_1, p_1)$
$(n_3 := n_1)$
$(n_1, m_1, o_1, p_1 := n_1, m_1, 4, p_1)$
$(n_5, m_5 := n_2, m_2)$
$(n_5, m_5 := m_2, n_2)$
$(m_1, n_1, o_1, p_1 := n_3, n_3, 4, n_3 + 1)$
$(n_4, m_4, o_4 := n_4, m_4 + 1, n_4)$
$(n_3 := n_4)$
$(n_3 := n_5)$
$\}$
$E = \{$
$(X_1, (n_2, m_2 := o_1, p_1), X_2),$
$(X_1, (n_3 := n_1), X_3),$
$(X_1, (n_1, m_1, o_1, p_1 := n_1, m_1, 4, p_1), X_1),$
$(X_2, (n_5, m_5 := n_2, m_2), X_5),$
$(X_2, (n_5, m_5 := m_2, n_2), X_5),$
$(X_3, (m_1, n_1, o_1, p_1 := n_3, n_3, 4, n_3 + 1), X_1),$
$(X_4, (n_4, m_4, o_4 := n_4, m_4 + 1, n_4), X_4),$
$(X_4, (n_3 := n_4), X_3),$
$(X_5, (n_3 := n_5), X_3)$
$\}$

Figure 4.1 shows the resulting graph.

When evaluating the assignments on a dependency graph we only have information about the parameters, the local variables are not known in this scope. Local variables are introduced by $\forall$ and the $\exists$ quantifications. It is usually not the case that an argument that contains variables bound by quantifications are constant. This is why we replace a term $(\overrightarrow{e})_i$ of predicate variable

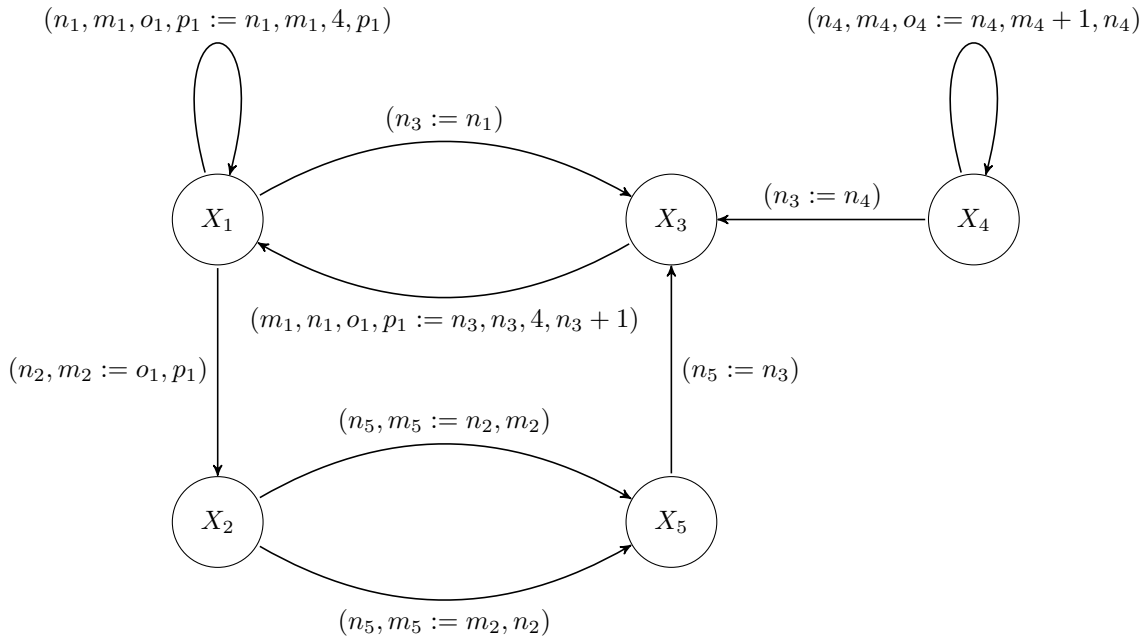



Figure 4.1: Dependency graph of Example 4.1.3

$X(\overrightarrow{e})$ to the value $NaC$ if $(\overrightarrow{e})_i$ contains a local variable. Here $NaC$ stands for *not a constant*. If we have a set of local variables we can use the function *RemoveLocal*, as defined below, to remove local variables from assignments.

**Definition 4.1.4.** For assignment $\overrightarrow{d} := \overrightarrow{e}$ the function $RemoveLocal(local, \overrightarrow{e})$, where *local* is the set of local variables, is defined as:

$$(\overrightarrow{e})_i := \begin{cases} (\overrightarrow{e})_i & if \quad FV((\overrightarrow{e})_i) \cap local = \emptyset \\ NaC & if \quad FV((\overrightarrow{e})_i) \cap local \neq \emptyset \end{cases}$$

The set of local variables of predicate formula $\varphi$ of equation $\sigma X(\overrightarrow{d} : \overrightarrow{D}) = \varphi$ is $FV(\varphi) \setminus \overrightarrow{d}$

Using the dependency graph we know how parameters are affected by other equations, moreover how some parameters affect other parameters. We now need a way to keep track which parameters are assumed to be constant, which parameters are not constant and for which parameters we do not know it yet. Here we encode these three possibilities by adding assertions:

| | |
|---|---|
| Parameter $n$ is assumed to be constant | assertion $\{n = v\}$, where $v$ is a constant value, is present |
| Parameter $n$ is not constant | assertion $\{n = NaC\}$, where $NaC$ is short for *Not a Constant*, is present |
| It is unknown if parameter $n$ is constant or not | No assertion about parameter $n$ is present |

There can be at most one assertion per parameter since a parameter cannot have multiple con-

stant values. The predicate $\kappa$ is used to obtain the initial assertions. Every predicate variable in $\kappa$ instantiates parameters of the corresponding PBES. These instantiations are the initial assertions. For example for $\kappa = X_4(1, 2, 3)$ with the PBES of example 4.1.3, the initial assertions are $\{n_4 = 1\}, \{m_4 = 2\}$ and $\{o_4 = 3\}$. It can be the case that a predicate $\kappa$ instantiates a predicate variable with two different values, e.g. $\kappa = X(0) \wedge X(1)$ for PBES $\mu X(n : \mathbb{N}) = n > 2$. In this case parameter $n$ cannot be a constant, thus the assertion $\{n = NaC\}$ is the initial assertion.
The assertions can now be used to calculate values of the right hand side of the assignments of the edges of the dependency graph. Not all edges have to be calculated, only the edges starting from vertices for which assertions are added. In the previous example, only edges starting from $X_4$ have to be calculated since only the parameters of $X_4$ have an assertion.

We now present an analysis of what happens if we investigate edge $\{X_i, d_j := d_i, X_j\}$ and have the assertion $\{d_i = v\}$ for constant $v$. We consider the following three cases for $d_j$:

1. No assertion for $d_j$ is present; this means it is not yet known if $d_j$ is a constant or a non constant. After the assignment $d_j := d_i$, the assertion $\{d_i = v\}$ must be added.
2. There already is an assertion for $d_j$, namely $\{d_j = w\}$, where $w$ is a constant. If:
   - $v = w$: after the assignment $d_j := d_i$ the existing assertion is still correct.
   - $v \neq w$: after the assignment $d_j := d_i$ the assertion $\{d_j = w\}$ is wrong because $d_j$ is assigned a different value than $w$. This means $d_j$ is not a constant and the assertion $\{d_j = w\}$ must be replaced by $\{d_j = NaC\}$.
3. There already is an assertion for $d_j$, namely $\{d_j = NaC\}$. This assertion can never change.

Note that during this analysis new assertions are added, and old assertions are replaced. We need to check if this new set of assertions is valid and detect if more assertions can be added. This is done by inspecting the edges again. Not all edges have to be inspected, only those edges starting from an equation of which the parameters have new assertions. This can be done with the help of the dependency graph. Using this analysis we can derive an algorithm that detects constant parameters:

We will now describe an algorithm for finding constants using predicate $\kappa$ and PBES $\mathcal{E}$. For this we will use a set $S$, which is the set of vertices that has to be inspected. This set is initially empty.

1. $G = (V, E, L)$ = dependency graph of $\mathcal{E}$
2. For every predicate variable instantiation $X_i(v)$ in $\kappa$, where $v$ is a constant value:
   - If there is no assertion for $d_i$ yet, add assertion $\{d_i = v\}$ and add $X_i$ to global set $S$.
   - If there is already is an assertion $\{d_i = w\}$ (because predicate variable $X_i$ occurs multiple times in $\kappa$) *and* $v \neq w$, replace $\{d_i = w\}$ by $\{d_i = NaC\}$. Note that $X_i$ was already placed in the set $S$ when assertion $\{d_i = w\}$ was added.
3. While $S$ is not empty do:
   (a) Take a vertex $X_i$ from $S$, and remove $X_i$ from $S$.
   (b) For all edges $(X_i, l, X_j)$, use the assertions to calculate the right hand side of assignment $l$.
   (c) For all parameters of $X_j$:
      - If the value of the assignment to parameter $d_j$ cannot be calculated because it depends on an unknown value:
        - If there was an assertion $\{d_j = v\}$ present, replace the assertion $\{d_j = v\}$ by $\{d_j = NaC\}$ and add $X_j$ to $S$.
        - If there was no assertion for $d_j$ present, add the assertion $\{d_j = NaC\}$ and add $X_j$ to $S$.
      - If the value $v$ of the assignment to parameter $d_j$ can be calculated:
        - If $v$ invalidates an assertion $\{d_j = w\}$, replace $\{d_j = w\}$ by $\{d_j = NaC\}$ and add $X_j$ to $S$
        - If there is no assertion for $d_j$, add the assertion $\{d_j = v\}$ and add $X_j$ to $S$.

In the first step the graph is constructed using Definition 4.1.2. The second step initializes the algorithm. Step three propagates known values and detects constants and non constants. If an assertion is added or replaced for a parameter of vertex $X_i$, this vertex has to be investigated again, so it is put back in the global set $S$.
Every iteration a vertex is removed from $S$, however, also multiple vertices can be added to $S$. Still this algorithm always terminates. This is true because a vertex is only added when an assertion changes. An assertion for parameter $d$ can change from:

- no assertion for $d$ to assertion $\{d = v\}$ for constant $v$.
- no assertion for $d$ to assertion $\{d = NaC\}$.
- assertion $\{d = v\}$ for constant $v$ to $\{d = NaC\}$.

Here it is clear that an assertion can change at most two times, i.e. from no assertion to $\{d = v\}$ to $\{d = NaC\}$. This means the iteration can only be executed # parameters $\times 2$ times.

After the algorithm finishes, some parameters have an assertion and some do not. Here we discuss what to do in such a case.

| | |
|---|---|
| There is an assertion $\{d = v\}$ for parameter $d$ | Parameter $d$ is a constant for the calculation of predicate variables $\kappa$. All occurrences of parameter $d$ in the corresponding predicate formula may be replaced by constant $v$. Since parameter $d$ does not occur in the predicate formula anymore it can be removed from the equation. This can be done with using Definition 3.1.11. |
| There is an assertion $\{d = NaC\}$ for parameter $d$ | Parameter $d$ is a non constant, it may not be replaced by any value |
| There is no assertion for parameter $d$ | Note that parameters of a certain equation for predicate variable $X$ only get assertions when $X$ can be reached in the dependency graph from a node in the set $S$. The fact that parameter $d$ belonging to the equation for a predicate variable $X_i$ does not have an assertion means that the equation for predicate variable $X_i$ cannot be reached. This means that the equation for predicate $X_i$ is not influential *for the calculation of* $\kappa$ and may be removed. |

**Example 4.1.5.** Here we discuss how we can use the algorithm presented above to find constants of the PBES presented in Example 4.1.3 if we want to know the solution of predicate $\kappa = X_4(0, 0, 0)$. For this the dependency graph which was already created in Example 4.1.3 will be used. In step two of our algorithm three assertions are added: $\{n_4 = 0\}, \{m_4 = 0\}$ and $\{o_4 = 0\}$. Finally vertex $X_4$ is put in the global set $S$. Here we present the bookkeeping for the initialization phase:

$S = \{X_4\}$
Assertions: $\{n_4 = 0\}, \{m_4 = 0\}, \{o_4 = 0\}$

The iteration step removes $X_4$ from $S$ and the values of the right hand side of $l$ of all outgoing edges of $X_4$ are calculated. This results in $(X_4, (n_4, m_4, o_4 := 0, 1, 0), X_4)$ and $(X_4, (n_3 := 0), X_3)$. According to the algorithm, the assertion $\{m_4 = 0\}$ is replaced by $\{m_4 = NaC\}$, the assertion $\{n_3 = 0\}$ is added, and $X_4$ and $X_3$ are added to the set $S$. We can continue the iteration step until $S$ is empty. The bookkeeping when the algorithm finishes is presented below.

$S = \emptyset$
Assertions: $\{n_1 = NaC\}, \{m_1 = NaC\}, \{o_1 = 4\}, \{p_1 = NaC\},$
$\{n_2 = 4\}, \{m_2 = NaC\},$
$\{n_3 = NaC\},$
$\{n_4 = 0\}, \{m_4 = NaC\}, \{o_4 = 0\},$
$\{n_5 = NaC\}, \{m_5 = NaC\}$

From the list of assertions we can read that $X_1$ has one constant, $X_2$ has one constant and $X_4$ has two constants. The constants from the assertions can be substituted for the corresponding parameters. When we do this we obtain the following PBES:

$$\begin{aligned}
\mu X_1(n_1, m_1, p_1 : \mathbb{N}) &= (n_1 \leq m_1 \vee X_2(4, p_1)) \wedge X_3(n_1) \wedge X_1(n_1, m_1, 4, p_1) \\
\mu X_2(m_2 : \mathbb{N}) &= X_5(4, m_2) \vee X_5(m_2, 4) \\
\nu X_3(n_3 : \mathbb{N}) &= n_3 \leq 3 \vee X_1(n_3, n_3, 4, n_3 + 1) \\
\nu X_4(m_4 : \mathbb{N}) &= 0 \leq (m_4 + 0) \vee (X_3(0) \wedge X_4(0, m_4 + 1, 0)) \\
\mu X_5(n_5, m_5 : \mathbb{N}) &= n_5 > m_5 \vee X_3(n_5)
\end{aligned}$$

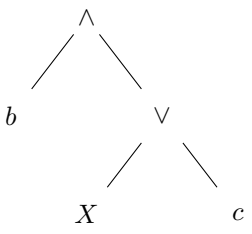
Note that now data term $0 \leq (m_4 + 0)$ of the equation for $X_4$ can be reduced to *true*. This means the whole right hand side of the equation for $X_4$ can be simplified to *true*.
In the next section we use the ability to calculate the value of data terms using the assertions to present a *ConstElm* algorithm that can find more constant values.

### 4.1.2 Adding conditions

The algorithm described in Section 4.1.1 finds constants by inspecting the dependency graph. The dependency graph does not consider that some assignments do not influence the solution of a PBES. For example, if we want to know the truth of predicate $X(5)$, given that $X$ is defined by the PBES $\mu X(n : \mathbb{N}) = (n > 3 \vee X(n + 1)) \wedge X(n)$, substituting 5 for parameter $n$ in this PBES, results in $\mu X(n : \mathbb{N}) = (5 > 3 \vee X(6)) \wedge X(5)$. Here $n$ is not constant, but simplifying the right hand side yields $\mu X(n : \mathbb{N}) = X(5)$, since $5 > 3$ yields *true* and $true \vee X(6)$ yields *true*. In this example $n$ is constant, but our algorithm will not mark $n$ as such. The reason that $n$ is not marked as a constant is because of the predicate variable instantiation $X(n + 1)$. This predicate variable instantiation results in the edge $(X, (n := n + 1), X)$ in the dependency graph of the PBES. This edge will mark $n$ as $NaC$. When $n > 3$ holds, predicate variable $X(6)$ is not influential; the predicate formula $(n > 3 \vee X(6)) \wedge X(5)$ can be rewritten to a form where predicate variable $X(6)$ is no longer present. This means that predicate variable instantiation $X(6)$ only is influential, if $n \leq 3$. On a dependency graph level, edge $(X, (n := n + 1), X)$ only has to be inspected if $n \leq 3$. Here $n \leq 3$ is a condition that can be placed on the edge $(X, (n := n+1), X)$. In the next sections we present a way to calculate the conditions that can be placed on edges. When we use the algorithm of the previous section, but do not consider edges in step 3b where the condition can be reduced to $false$, more constants can be found. When the improved algorithm is applied on the example presented above, $n$ will be marked as constant.

By the structure of a predicate formula, we can represent the formula as a tree. This representation can enhance readability and can make reasoning about the formula easier. In the following sections we will use both the tree representation and the textual representation. For example, we will use both the following notations:

$b \wedge (X \vee c)$



#### 4.1.2.1 True/false conditions

We usually cannot tell whether a predicate variable evaluates to $true$ or $false$ before calculating the whole PBES. Data terms on the other hand can be evaluated when parameter values are known. Therefore, determining if a predicate formula $\varphi$ is $true$ or $false$ is done by inspecting data terms. For example, if $\varphi = b_1 \wedge b_2$, $\varphi$ is only true if $b_1 \wedge b_2$ holds, and $\varphi$ is $false$ when $\neg b_1 \vee \neg b_2$ holds. In this case condition "$b_1 \wedge b_2$" is the *negation* of "$\neg b_1 \vee \neg b_2$". Unfortunately this is not always the case because we do not know the value of predicate variables; if we have $\varphi = b \wedge Y(\overrightarrow{e})$, we know $\varphi$ is $false$ if $\neg b$ holds, but cannot determine when $\varphi$ is $true$ without evaluating the predicate variable $Y(\overrightarrow{e})$. Therefore we keep track of a *true-condition and* a *false-condition*, that state what should hold for a predicate formula to evaluate to $true$ and $false$ respectively.
The following observation explains why true/false conditions are necessary: for predicate variable instantiation $X(e) \in PV(\varphi)$ and, using the assertions discussed in Section 4.1.1, if we know that the *true-condition* or *false-condition* holds, then $X(e)$ is *not* influential. This is true because $\varphi$ can be reduced to *true* if its *true-condition* holds, and $\varphi$ can be reduced to $false$ if its *false-condition* holds. Next we define what the *true-condition* and *false-condition* for a predicate formulas are.

**Definition 4.1.6.** The *true-condition* and *false-condition* of a predicate formula $\varphi$ are calculated with the functions $TC(\varphi)$ and $FC(\varphi)$, defined as:

$$
\begin{array}{ll}
TC(b) & = b \\
FC(b) & = \neg b \\
TC(X(e)) & = false \\
FC(X(e)) & = false \\
TC(\varphi_1 \wedge \varphi_2) & = TC(\varphi_1) \wedge TC(\varphi_2) \\
FC(\varphi_1 \wedge \varphi_2) & = FC(\varphi_1) \vee FC(\varphi_2) \\
TC(\varphi_1 \vee \varphi_2) & = TC(\varphi_1) \vee TC(\varphi_2) \\
FC(\varphi_1 \vee \varphi_2) & = FC(\varphi_1) \wedge FC(\varphi_2) \\
TC(\forall_{d:D}.\varphi) & = \forall_{d:D}.TC(\varphi) \\
FC(\forall_{d:D}.\varphi) & = \exists_{d:D}.FC(\varphi) \\
TC(\exists_{d:D}.\varphi) & = \exists_{d:D}.TC(\varphi) \\
FC(\exists_{d:D}.\varphi) & = \forall_{d:D}.FC(\varphi)
\end{array}
$$

**Property 4.1.7.** *The functions $TC(\varphi)$ and $FC(\varphi)$ have the following properties:*

1. $TC(\varphi) \rightarrow (\varphi)$
   $\varphi \rightarrow \neg FC(\varphi)$
2. *$TC(\varphi)$ and $FC(\varphi)$ do not contain predicate variables*
3. *$TC(\varphi)$ and $FC(\varphi)$ are the* weakest *functions for which property (1) and (2) holds.*

*Proof.*
(1) The proof of this property is straightforward using induction on the structure of a PBES. here we will only present the proof of $TC(\varphi_1 \wedge \varphi_2) \rightarrow (\varphi_1 \wedge \varphi_2)$

Induction Hypothesis: $TC(\varphi_1) \rightarrow (\varphi_1) \wedge TC(\varphi_2) \rightarrow (\varphi_2)$

$TC(\varphi_1 \wedge \varphi_2) \rightarrow (\varphi_1 \wedge \varphi_2)$
$=$ {Definition of $TC(\varphi)$}
$TC(\varphi_1) \wedge TC(\varphi_2) \rightarrow (\varphi_1 \wedge \varphi_2)$
$\Leftarrow$ {Induction Hypothesis}
$\varphi_1 \wedge \varphi_2 \rightarrow (\varphi_1 \wedge \varphi_2)$
$=$ {logic}
true

(2) The proof of this property again can be proved using induction on the structure of a PBES. By looking at the base cases it can also be seen that no predicate variables are present in the conditions.
(3) We will only prove this for $TC(\varphi)$, the proof for the $FC(\varphi)$ is similar. Suppose there is a weaker function $\mathcal{T}(\varphi)$ for which (1) and (2) holds. Then we can show this function $\mathcal{T}(\varphi) = TC(\varphi)$, for all environments $\eta$ and $\varepsilon$:
$(\llbracket \mathcal{T}(\varphi) \rightarrow \varphi \rrbracket \eta\varepsilon \wedge \llbracket TC(\varphi) \rightarrow \mathcal{T}(\varphi) \rrbracket \varepsilon) \rightarrow \llbracket \mathcal{T}(\varphi) \rrbracket \varepsilon = \llbracket TC(\varphi) \rrbracket \varepsilon$

case $b$:

$\llbracket \mathcal{T}(b) \rrbracket \eta\varepsilon \rightarrow \llbracket b \rrbracket \eta\varepsilon$
$=$ {Semantics, $\mathcal{T}$ contains no predicate variables}
$\llbracket \mathcal{T}(b) \rrbracket \varepsilon \rightarrow \llbracket b \rrbracket \varepsilon$

Solutions for $\llbracket \mathcal{T}(b) \rrbracket \varepsilon$ are: $false, b$
The weakest function $\llbracket \mathcal{T}(b) \rrbracket \varepsilon = b = TC(b)$

case $X$:

$\llbracket \mathcal{T}(X(e)) \rrbracket \eta\varepsilon \rightarrow \llbracket X(e) \rrbracket \eta\varepsilon$
$=$ {Semantics, $\mathcal{T}$ contains no predicate variables}
$\llbracket \mathcal{T}(X(e)) \rrbracket \varepsilon \rightarrow (\eta(X))(\llbracket e \rrbracket \varepsilon)$
$\Rightarrow$ {Take $\eta(X) = \lambda_{v:D}.false$}
$\llbracket \mathcal{T}(X(e)) \rrbracket \varepsilon \rightarrow false$

Solutions for $\llbracket \mathcal{T}(X(e)) \rrbracket \varepsilon$ are: $false$
The weakest function $\llbracket \mathcal{T}(X(e)) \rrbracket \varepsilon = false = TC(b)$

Using induction hypothesis:
$(\llbracket \mathcal{T}(\varphi_i) \rightarrow \varphi_i \rrbracket \eta\varepsilon \wedge \llbracket TC(\varphi_i) \rightarrow \mathcal{T}(\varphi_i) \rrbracket \varepsilon) \rightarrow \llbracket \mathcal{T}(\varphi_i) \rrbracket \varepsilon = \llbracket TC(\varphi_i) \rrbracket \varepsilon$

case $\varphi_1 \wedge \varphi_2$:

$\llbracket \mathcal{T}(\varphi_1 \wedge \varphi_2) \rrbracket \eta\varepsilon \rightarrow \llbracket \varphi_1 \wedge \varphi_2 \rrbracket \eta\varepsilon$
$=$ {Semantics, $\mathcal{T}$ contains no predicate variables, logic}
$(\llbracket \mathcal{T}(\varphi_1 \wedge \varphi_2) \rrbracket \varepsilon \rightarrow (\llbracket \varphi_1 \rrbracket \eta\varepsilon) \wedge (\llbracket \mathcal{T}(\varphi_1 \wedge \varphi_2) \rrbracket \varepsilon \rightarrow (\llbracket \varphi_2 \rrbracket \eta\varepsilon)$

From $[\![TC(\varphi_1 \wedge \varphi_2) \rightarrow \mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon$ we can derive:
$([\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon) \vee ([\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon)$
Case distinction:

- Case $[\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon \wedge \neg([\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon)$
  From the induction hypothesis it holds that $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon$.
  Since we have $\neg([\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon) = [\![TC(\varphi_2)]\!]\varepsilon \wedge \neg[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon$ and $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon$, we may conclude that $[\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![TC(\varphi_2)]\!]\varepsilon$. This means we have: $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon \wedge [\![TC(\varphi_2)]\!]\varepsilon$
- Case $[\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon \wedge \neg([\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon)$
  Symmetrically
- Case $[\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon \wedge [\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon$
  Induction hypothesis: $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon$
  Induction hypothesis: $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_2)]\!]\varepsilon$
  Hence: $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon \wedge [\![TC(\varphi_2)]\!]\varepsilon$

Conclusion: $[\![\mathcal{T}(\varphi_1 \wedge \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon \wedge [\![TC(\varphi_2)]\!]\varepsilon$

case $\varphi_1 \vee \varphi_2$:

$$
\begin{array}{ll}
 & [\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\eta\varepsilon \rightarrow [\![\varphi_1 \vee \varphi_2]\!]\eta\varepsilon \\
= & \quad \{\text{Semantics, } \mathcal{T} \text{ contains no predicate variables, logic}\} \\
 & ([\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon \rightarrow ([\![\varphi_1]\!]\eta\varepsilon) \vee ([\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon \rightarrow ([\![\varphi_2]\!]\eta\varepsilon)
\end{array}
$$

From $[\![TC(\varphi_1 \vee \varphi_2) \rightarrow \mathcal{T}(\varphi_1 \vee \varphi_2]\!]\varepsilon$ we can derive:
$([\![TC(\varphi_1)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon) \wedge ([\![TC(\varphi_2)]\!]\varepsilon \rightarrow [\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon)$

We will invest this case:

- Induction hypothesis: $[\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon$
  Induction hypothesis: $[\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon = [\![TC(\varphi_2)]\!]\varepsilon$
  Hence: $[\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon \vee [\![TC(\varphi_2)]\!]\varepsilon$

Conclusion: $[\![\mathcal{T}(\varphi_1 \vee \varphi_2)]\!]\varepsilon = [\![TC(\varphi_1)]\!]\varepsilon \vee [\![TC(\varphi_2)]\!]\varepsilon$

Case $Q_{d:D}.\varphi_1$:

$$
\begin{array}{ll}
 & [\![\mathcal{T}(Q_{d:D}.\varphi_1)]\!]\eta\varepsilon \rightarrow [\![Q_{d:D}.\varphi_1]\!]\eta\varepsilon \\
= & \quad \{\text{Semantics, } \mathcal{T} \text{ contains no predicate variables, logic}\} \\
 & ([\![\mathcal{T}(Q_{d:D}.\varphi_1)]\!]\varepsilon \rightarrow Q_{v:D}.[\![\varphi_1]\!]\eta\varepsilon[v/d])
\end{array}
$$

From $[\![Q_{d:D}.TC(\varphi_1) \to \mathcal{T}(Q_{d:D}.\varphi_1)]\!]\varepsilon$ we can derive:
$[\![Q_{d:D}.TC(\varphi_1)]\!]\varepsilon \to [\![\mathcal{T}(Q_{d:D}.\varphi_1)]\!]\varepsilon$

We will invest this case:

- Induction hypothesis: $[\![Q_{d:D}.TC(\varphi_1)]\!]\varepsilon = [\![\mathcal{T}(Q_{d:D}.\varphi_1)]\!]\varepsilon$

Conclusion: $[\![Q_{d:D}.TC(\varphi_1)]\!]\varepsilon = [\![\mathcal{T}(Q_{d:D}.\varphi_1)]\!]\varepsilon$ □

To keep track of which true/false conditions hold at a certain point in the tree of a predicate formula, the true/false conditions are placed on nodes of the predicate formula. For node $\otimes$, we write the true/false conditions to this node as $\otimes^{TC(\otimes)}_{FC(\otimes)}$. Now the *true-condition* on a node states what condition should hold to make the whole subtree of the node *true*, and the *false-condition* states what condition should hold to make the whole subtree of the node $false$. For readability purposes we do not write true/false conditions on data terms and predicate variables. These conditions can be read directly from the data term/predicate variable using Definition 4.1.6.

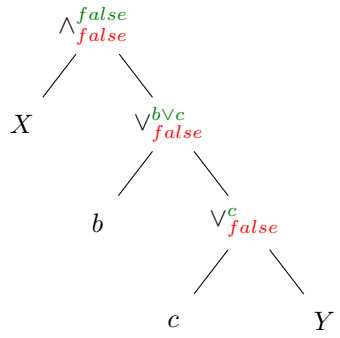
**Example 4.1.8.** Here we show how to use the functions $TC(\varphi)$ and $FC(\varphi)$ as defined in Definition 4.1.6 to add conditions to the nodes of predicate formula $\varphi = X \wedge (b \vee (c \vee Y))$. Here $X$ and $Y$ are predicate variables and $b$ and $c$ are data terms. We apply the definitions to get the following derivation:

$$
\begin{aligned}
& TC(X \wedge (b \vee (c \vee Y))) \\
= \quad & \{\text{def } TC(\varphi_1 \wedge \varphi_2)\} \\
& TC(X) \wedge TC((b \vee (c \vee Y))) \\
= \quad & \{\text{def } TC(X), \text{ def } TC(\varphi_1 \vee \varphi_2)\} \\
& false \wedge (TC(b) \vee TC((c \vee Y))) \\
= \quad & \{\text{def } TC(b), \text{ def } TC(\varphi_1 \vee \varphi_2)\} \\
& false \wedge (b \vee (TC(c) \vee TC(Y))) \\
= \quad & \{\text{def } TC(b), \text{ def } TC(X)\} \\
& false \wedge b \vee c \\
= \quad & \{\text{logic}\} \\
& false
\end{aligned}
$$

This derivation continued even though it is clear that $(false \wedge \ldots)$ would result in the condition $false$. This is done because now all *true-conditions* of all sub trees of $\varphi$ are known:

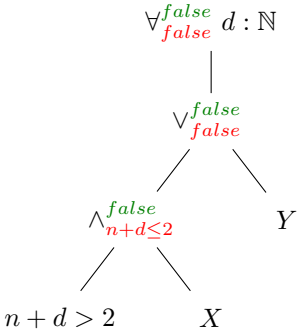
$$
\begin{aligned}
& TC(X \wedge (b \vee (c \vee Y))) = false \\
& TC(b \vee (c \vee Y)) = b \vee c \\
& TC(c \vee Y) = c
\end{aligned}
$$

The *false-conditions* can be calculated in a similar way using $FC(\varphi)$. When we place the true/false conditions on the corresponding nodes the result is:




When looking at the predicate formula $\varphi$, if there is a predicate variable $X$ in the subtree of $\varphi$, and $TC(\varphi)$ *or* $FC(\varphi)$ holds, $X$ is *not* influential. This means the predicate variable can only be influential if both the *true-condition* and the *false-condition* do not hold. Furthermore, if for example $\varphi = \varphi_1 \wedge \varphi_2$ and $X$ is in the subtree of $\varphi_2$, $X$ is *not* influential if either $TC(\varphi_2)$ or $FC(\varphi_2)$ holds. All nodes $\varphi$ on the path from the root to $X$ are nodes of which $X \in PV(\varphi)$. This means that in general, a predicate variables is not influential if any true/false condition holds on a node which is on the path from the root to that predicate variable.

True/false conditions consist of data terms, together with logical connectors. A data term can contain variables; such a variable is either a parameter or a variable bound by a quantification, which is higher up in the tree. The true/false conditions obtained using Definition 4.1.6 however, do not bind these variables. As an illustration take the following predicate formula: $\forall_{d:\mathbb{N}}.(n+d > 2 \wedge X) \vee Y$, where $n$ is a parameter of type $\mathbb{N}$. In the tree with true/false conditions added, $d$ not bound in the *false-condition* $n + d \leq 2$




This is solved by binding the data term in which $d$ occurs with a quantification. Similar like in Definition 4.1.6, for a universal quantification, the *true-condition* is prefixed with a universal quantification and the *false-condition* is prefixed with an existential quantification. For an existential quantification this is the other way around. In this case the new *false-condition* is $\exists_{d:\mathbb{N}}.(n+d \leq 2)$, $X$ is influential if $\neg(\exists_{d:\mathbb{N}}.(n+d \leq 2))$ holds. Later we will give a definition that adds quantifications to a set of conditions.

#### 4.1.2.2 *inf-condition*

In this section we will give the definition of the *inf-condition* for a certain predicate variable instantiation $X(e)$. An *inf-condition* is the condition that must hold for the predicate variable instantiation to be influential. When constructing edges for the dependency graph from the predicate variable instantiation, the *inf-condition* can be placed on the edges to indicate when the edge is influential. We start by giving a definition of a function that adds quantifications to a set of conditions. This will later be used to bind unbound variables in true/false conditions, as discussed in the previous section.

**Definition 4.1.9.** To add quantifications to elements of a set of true/false conditions $cs$, we introduce $ApplyQ(Q_{d:D}, cs)$ which is defined as:

$$ApplyQ(Q_{d:D}, cs) = \{Q_{d:D}.c | c \in cs\}$$

Using Definition 4.1.9 we can collect all *true-conditions* and *false-conditions* for a predicate variable instantiation $X_i(e)$ in predicate formula $\varphi$, without having free variables. This is done by collecting all conditions from the root of $\varphi$ to $X_i(e)$. Here we assume that predicate variable instantiations are unique in a certain predicate formula $\varphi$. The algorithm in Appendix B provides an algorithm without this constraint. This is done by constructing the conditions of all predicate variable instantiations at the same time.

**Definition 4.1.10.** The set of all *true-conditions* of predicate variable instantiation $X_i(e)$ in predicate formula $\varphi$ is $Cond_T(X_i(e), \varphi)$, where $Cond_T(X_i(e), \varphi)$ is defined as:

$$
\begin{array}{ll}
Cond_T(X_i(e), b) & = \emptyset \\
Cond_T(X_i(e), X_j(k)) & = \emptyset \\
Cond_T(X_i(e), \varphi_1 \oplus \varphi_2) & = \\
\quad \left\{ \begin{array}{ll} TC(\varphi_1 \oplus \varphi_2) \cup Cond_T(X_i(e), \varphi_1) & if\ X_i(e) \in PV(\varphi_1) \\ TC(\varphi_1 \oplus \varphi_2) \cup Cond_T(X_i(e), \varphi_2) & if\ X_i(e) \in PV(\varphi_2) \\ \emptyset & otherwise \end{array} \right. & \\
Cond_T(X_i(e), \forall_{d:D}.\varphi) & = \\
\quad \left\{ \begin{array}{ll} TC(\forall_{d:D}.\varphi) \cup ApplyQ(\forall_{d:D}, Cond_T(X_i(e), \varphi)) & if\ X_i(e) \in PV(\varphi) \\ \emptyset & otherwise \end{array} \right. & \\
Cond_T(X_i(e), \exists_{d:D}.\varphi) & = \\
\quad \left\{ \begin{array}{ll} TC(\exists_{d:D}.\varphi) \cup ApplyQ(\exists_{d:D}, Cond_T(X_i(e), \varphi)) & if\ X_i(e) \in PV(\varphi) \\ \emptyset & otherwise \end{array} \right. &
\end{array}
$$

The set of all *false-conditions* of predicate variable instantiation $X_i(e)$ in predicate formula $\varphi$ is $Cond_F(X_i(e), \varphi)$, where $Cond_F(X_i(e), \varphi)$ is defined as:

$$
\begin{array}{ll}
Cond_F(X_i(e), b) & = \emptyset \\
Cond_F(X_i(e), X_j(k)) & = \emptyset \\
Cond_F(X_i(e), \varphi_1 \oplus \varphi_2) & = \\
\quad \left\{ \begin{array}{ll} FC(\varphi_1 \oplus \varphi_2) \cup Cond_F(X_i(e), \varphi_1) & if\ X_i(e) \in PV(\varphi_1) \\ FC(\varphi_1 \oplus \varphi_2) \cup Cond_F(X_i(e), \varphi_2) & X_i(e) \in PV(\varphi_2) \\ \emptyset & otherwise \end{array} \right. &
\end{array}
$$

$$Cond_F(X_i(e), \forall_{d:D}.\varphi) = \begin{cases} FC(\forall_{d:D}.\varphi) \cup ApplyQ(\exists_{d:D}, Cond_F(X_i(e), \varphi)) & if\ X_i(e) \in PV(\varphi) \\ \emptyset & otherwise \end{cases}$$

$$Cond_F(X_i(e), \exists_{d:D}.\varphi) = \begin{cases} FC(\exists_{d:D}.\varphi) \cup ApplyQ(\forall_{d:D}, Cond_F(X_i(e), \varphi)) & if\ X_i(e) \in PV(\varphi) \\ \emptyset & otherwise \end{cases}$$

Using $Cond_T(X_i(e), \varphi)$ and $Cond_F(X_i(e), \varphi)$ we can collect all true/false conditions for a predicate variable in a certain predicate formula. A predicate variable instantiation is only influential if the conjunction of the negation of all these conditions does not hold. Therefore we introduce a new definition.

**Definition 4.1.11.** For a set of true/false conditions $cs$, $Neg(cs)$ is defined as:

$$Neg(cs) = \bigwedge\{\neg c | c \in cs\}$$

Note that if $cs = \emptyset$, $Neg(cs) = true$

We can now give a formal definition of an *inf-condition*

**Definition 4.1.12.** *inf-condition* of predicate variable $X_i(e)$ in predicate formula $\varphi$ is defined as:

$$Neg(Cond_T(X_i(e), \varphi)) \wedge Neg(Cond_F(X_i(e), \varphi))$$

If the *inf-condition* for predicate variable $X_i(e)$ holds, $X_i(e)$ is influential.

*Note.* An *inf-condition* is expressed in predicate logic, which is in general undecidable. Therefore we cannot always evaluate the condition. If the *inf-condition* for predicate variable $X_i(e)$ does not hold, $X_i(e)$ is *not* influential. If the *inf-condition* for predicate variable $X_i(e)$ holds $X_i(e)$ is influential. If the *inf-condition* for predicate variable $X_i(e)$ cannot be reduced to $true$ or $false$ we consider $X_i(e)$ to be influential and do *not* skip the corresponding edge in the dependency graph. This is a safe approximation.

Now that the *inf-condition* is defined, we give an example of how to construct it.

**Example 4.1.13.** When true/false conditions are added to a predicate formula, the *inf-conditions* can be read from the formula. To illustrate this the tree with conditions of Example 4.1.8 is used to read *inf-condition* for $X$ and $Y$. For $X$ the only true/false-conditions on the path from the root to X is $false$, the negation of this is $true$. On the path from the root to $Y$ we encounter the true/false-conditions $(b \vee c)$ and $(c)$. The *inf-condition* of $Y$ is the conjunction of the negation of both, which is $\neg(b \vee c) \wedge \neg c$, which can be simplified to $\neg(b \vee c)$. When looking at the predicate formula we indeed see that if $b$ or $c$ holds, the expression $b \vee c \vee Y$ is always $true$ and $Y$ is thus only influential if its *inf-condition* holds.

When we have an *inf-condition* for every predicate variable instantiation, we can make a dependency graph and add these conditions. An edge of a dependency graph can now be written as

the four tuple $(X_i, (\overrightarrow{d}) := (\overrightarrow{e}), cond, X_j)$:

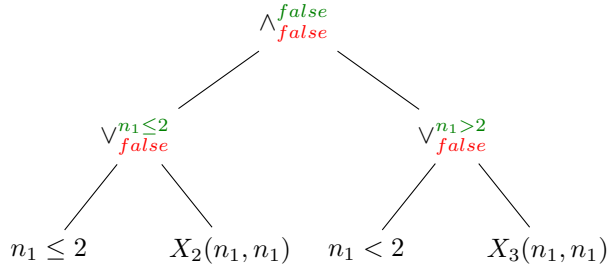
- $X_i \in V$ is the source of the edge.
- $\overrightarrow{d} := \overrightarrow{e}$ are the argument assignments.
- *cond* are the conditions that should hold before examining this edge.
- $X_j \in V$ is the destination of the edge.

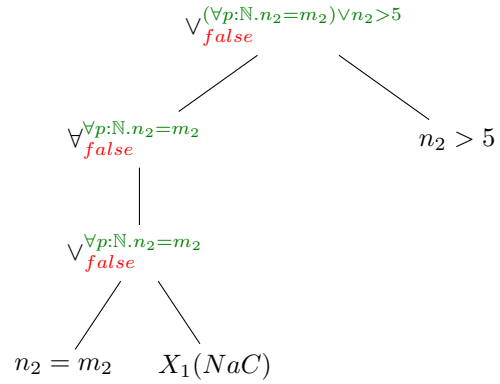
Now we give an example of how $ConstElm$ with conditions works.

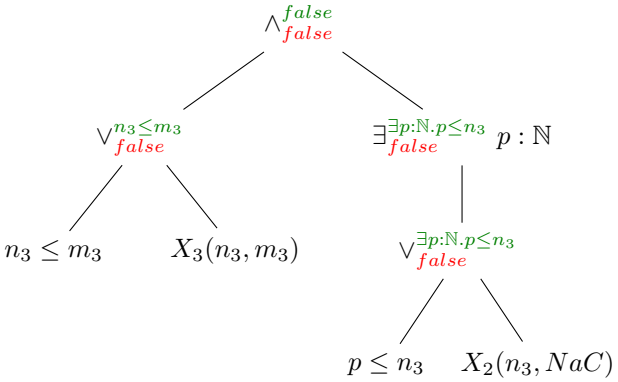
**Example 4.1.14.** Here we give an example of how the algorithm $ConstElm(\mathcal{E}, \kappa)$ works. Consider the following PBES:

$$\begin{array}{ll} \mu X_1(n_1 : \mathbb{N}) & = (n_1 \leq 2 \vee X_2(n_1, n_1)) \wedge (n_1 > 2 \vee X_3(n_1, n_1)) \\ \mu X_2(m_2, n_2 : \mathbb{N}) & = (\forall p : \mathbb{N}. n_2 = m_2 \vee X_1(p)) \vee n_2 > 5 \\ \nu X_3(m_3, n_3 : \mathbb{N}) & = (n_3 \leq m_3 \vee X_3(n_3, m_3)) \wedge (\exists p : \mathbb{N}. p \leq n_3 \vee X_2(n_3, p)) \end{array}$$

We first make a dependency graph and then discuss the result for different predicates $\kappa$. Using the definitions we can add *true-conditions* and *false-conditions* on the nodes. We also add quantifications to true/false-conditions of nodes below a quantification, and replace arguments with local variables to $NaC$. The result is shown below.



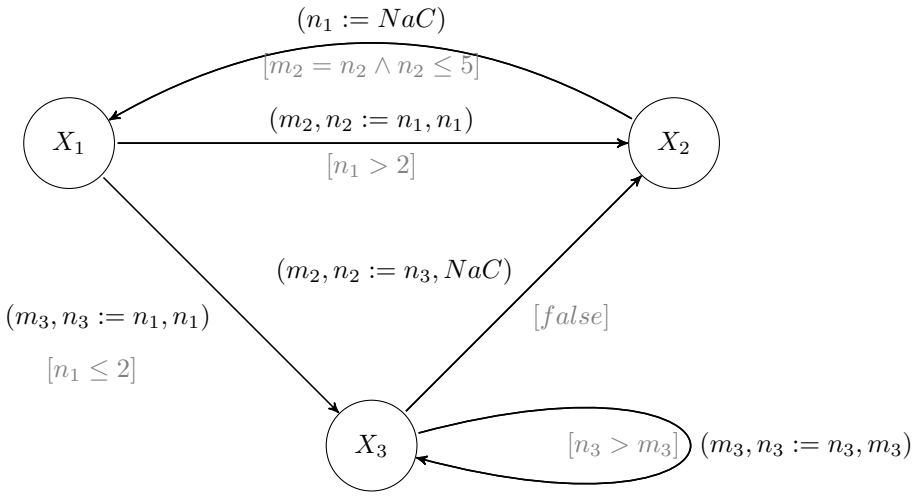




We can now read the *inf-conditions* for predicate variables and using these conditions we can make edges for the dependency graph. First we simplify the *inf-conditions*, then we add them to the edges of the dependency graph:

$$(X_1, (m_2, n_2 := n_1, n_1), n_1 > 2, X_2)$$
$$(X_1, (m_3, n_3 := n_1, n_1), n_1 \leq 2, X_3)$$
$$(X_2, (n_1 := NaC), (n_2 = m_2 \wedge n_2 \leq 5), X_1)$$
$$(X_3, (m_3, n_3 := n_3, m_3), n_3 > m_3, X_3)$$
$$(X_3, (m_2, n_2 := n_3, NaC), false, X_2)$$

We can now make a dependency graph with conditions. In the figure below, the conditions are added between square brackets.




In this graph, an edge is called enabled if its condition does not reduce to $false$ using the current assertions. We will now investigate some scenarios:

- Assume $\kappa = X_1(0)$, this means the assertion $\{n_1 = 0\}$ is added and vertex $X_1$ is marked for further inspection. The only valid edge now is $(X_1, (m_3, n_3 := n_1, n_1), n_1 \leq 2, X_3)$, which means the assertion $\{m_3 = 0\}$ and $\{n_3 = 0\}$ are added, and vertex $X_3$ is inspected further. Since vertex $X_3$ has no more valid edges the algorithm stops, the result is that the parameters of $X_1$ and $X_3$ are marked as constants with the value 0. The second equation can be removed.

- Assume $\kappa = X_1(10)$, this means that the assertion $\{n_1 = 10\}$ is added and vertex $X_1$ is marked for further inspection. After one iteration the assertions $\{m_2 = 10\}$ and $\{n_2 = 10\}$ are added and vertex $X_2$ is marked for further inspection. Since vertex $X_2$ has no more valid edges the algorithm stops. The constant parameters are the parameters of $X_1$ and $X_2$, their values are 10. The third equation can be removed.

- Assume $\kappa = X(3)$, this means that the assertion $\{n_1 = 3\}$ is added and vertex $X_1$ is marked for further inspection. After one iteration the assertions $\{m_2 = 3\}$ and $\{n_2 = 3\}$ are added and vertex $X_2$ is marked for further inspection. The next iteration the edge $(X_2, (n_1 := NaC), (n_2 = m_2 \wedge n_2 \leq 5), X_1)$ is valid, so the assertion $\{n_1 = 3\}$ is replaced by the assertion $\{n_1 = NaC\}$ and vertex $X_1$ is marked for further inspection. Since conditions with $NaC$ values cannot be reduced to $false$, the next iteration both the edge $(X_1, (m_2, n_2 := n_1, n_1), n_1 > 2, X_2)$ and $(X_1, (m_3, n_3 := n_1, n_1), n_1 \leq 2, X_3)$ are valid. When we continue the algorithm it ends in a state where all parameters are not constant.

## 4.2 Example using ConstElm

In this section we show the importance of *ConstElm* using the mCRL2 tool set. Here we apply *ConstElm* on the same PBESs as discussed in Section 3.3. These PBESs specify properties about the alternating bit protocol, which is presented in Appendix C. For completeness we repeat the four properties:
Property 1: No deadlock may occur

```
nu X. ([true]X && <true>true)
```

Property 2: A message that is sent can always eventually be received

```
nu X. ([true]X && forall d:Nat. [r1(d)] mu Y. (<true>Y || <s4(d)>true))
```

Property 3: The protocol does not generate new messages

```
forall d:Nat. nu X. (([!r1(d)]X && [s4(d)]false))
```

Property 4: The protocol does not duplicate messages

```
[true*] forall d:D. [r1(d).(!r1(d) && !s4(d))*.s4(d).(!r1(d))*.s4(d)]false
```

While creating a PBESs an intermediate LPS file is made. The mCRL2 tool kit has a tool called *lpsconstelm* which, as the name suggests, detects constants in the LPS file. In this case no constants are detected in the LPS file. Since a PBES only verifies a property of the ABP specification (i.e. not always the whole specification is used), it may detect more constants. We will present two tests, one test with only the algorithm as presented in Section 4.1, thus without the use of conditions, and one test which also takes conditions into account.

### 4.2.1 Algorithm without conditions

All tests are done on a windows machine with a 1.8Ghz Pentium M processor and 1Gb of RAM. The results are presented below:

| Property | Running time | Constants found | Parameters remaining | Type |
|---|---|---|---|---|
| 1 | 172 ms | 0 | 3 | Nat |
| | | 0 | 4 | Pos |
| | | 0 | 4 | Bool |
| 2 | 218 ms | 0 | 7 | Nat |
| | | 0 | 8 | Pos |
| | | 0 | 8 | Bool |
| 3 | 187 ms | 0 | 7 | Nat |
| | | 4 | 4 | Pos |
| | | 2 | 6 | Bool |
| 4 | 219 ms | 0 | 7 | Nat |
| | | 0 | 4 | Pos |
| | | 0 | 6 | Bool |

From this table we see that the running time of the algorithm is low. For the PBES encoding property three, the algorithm finds constants and the PBES can be simplified. This shows that

*ConstElm* is useful on a PBES level, even more because no constants were found on a LPS level. These simplifications usually (see Section 4.2.3) are not useful when using instantiation [DPWar] to solve a PBES. This is because instantiation also uses the information of $\kappa$. On the other hand, symbolic approximation will benefit from *ConstElm*. Since no stable implementation of this algorithm exists we cannot measure how much can be gained.

### 4.2.2 Algorithm with conditions

Here we present the results for *ConstElm* applied in the same PBESs as the previous section, now using the algorithm with conditions:

| Property | Running time | Constants found | Parameters remaining | Type |
|---|---|---|---|---|
| 1 | 1406 ms | 0 | 3 | Nat |
| | | 0 | 4 | Pos |
| | | 0 | 4 | Bool |
| 2 | 1718 ms | 0 | 7 | Nat |
| | | 0 | 8 | Pos |
| | | 0 | 8 | Bool |
| 3 | 1125 ms | 0 | 7 | Nat |
| | | 4 | 4 | Pos |
| | | 2 | 6 | Bool |
| 4 | 1765 ms | 0 | 7 | Nat |
| | | 0 | 4 | Pos |
| | | 0 | 6 | Bool |

Although *ConstElm* with conditions can detect more parameters in theory, it does not do so with for the four PBESs used here. This while the running time of the algorithm has increased. The extra time needed is mainly used to (try to) rewrite conditions. Unfortunately, for the systems tested, *ConstElm* with conditions cannot find more constants than *ConstElm* without conditions. Only with specially constructed test PBESs, *ConstElm* with conditions removes more parameters than *ConstElm* without conditions. An example of such a constructed PBES is:

$\mu X(n : \mathbb{N}) = 2 < n \vee X(n + 1)$
$\kappa = X(5)$

Here *ConstElm* with conditions detects parameter $n$ as constant. *ConstElm* without conditions does not detect the constant parameter $n$.

### 4.2.3 *ConstElm* combined with *ParElm*

*ParElm* detects parameters that do not influence the solution of a PBES and *ConstElm* uses information of a predicate $\kappa$ which we want to check to remove parameters. We can apply both algorithms (if $\kappa$ is known) on a PBES, but the order does matter. If *ConstElm* is applied first, constant parameters are removed *and* the parameters in the predicate formula $\varphi$ are replaced by constant values. These constant values may lead to the removal of occurrences of parameters in $\varphi$ if the PBES is simplified using logic rules. To illustrate this we present an example.
Suppose we want to check property $\kappa = X(0, true)$ for PBES:

$$\mu X(n : \mathbb{N}, b : \mathbb{B}) = (n > 5 \wedge b) \vee X(n, \neg b)$$

If the algorithm $ParElm$ is used first, none of the parameters are removes because $n$ and $b$ both occur in a data term. When the algorithm $ConstElm$ is used first, $n$ is marked as constant with value 0. The resulting PBES is:

$$\mu X(b : \mathbb{B}) = (0 > 5 \wedge b) \vee X(\neg b)$$

This PBES can be simplified to:

$$\mu X(b : \mathbb{B}) = X(\neg b)$$

Now $ParElm$ can remove the parameter $b$ which results in the simple PBES:

$$\mu X = X$$

The reason that this works is because $ConstElm$ may change the solution of a PBES as long as the solution of $\kappa$ does not change. The altered PBES sometimes can be simplified in such a way that parameters in data terms are removed.

However, running $ParElm$ after $ConstElm$ only removed more parameters in specially constructed test cases. Running $ConstElm$ after $ParElm$ will never lead to the detection of more constants by $ConstElm$, since $ParElm$ produces an equal PBES (modulo typing).

# Chapter 5

# Conclusion

In this thesis we have presented techniques that can be used to simplify PBESs. These techniques work on the syntax of a PBES, so they can be applied before the calculation of the solution of a PBES. The presented techniques can be divided into two parts; techniques to identify superfluous parameters and techniques to identify constant parameters in PBESs.

The first technique, presented in Chapter 3, transforms the problem of finding superfluous parameters to a reachability problem. A method called instantiation [DPWar], which can be used to solve PBESs, can benefit when superfluous parameters are removed. This is tested with the mCRL2 toolkit which has, among other useful tools, a tool that solves PBESs using instantiation. This tool is called *pbes2bool*. Section 3.3 even presents a PBES which could not be solved by instantiation before parameters are removed, but can be solved after superfluous parameters are removed.

The second technique uses some extra information to detect constant values in a PBES. In the case a PBES encodes a modal checking problem or an equivalence problem, this extra information is provided by the initial state of the involved process. The technique called symbolic approximation [GW04] can benefit when constant values are detected. As no stable implementation of a tool using symbolic approximation exists yet, we cannot tell how much can be gained when parameters are replaced by constants. Instantiation can, indirectly, benefit from these techniques too, as explained in Section 4.2.3.

From the techniques presented in this thesis, two tools have been constructed and added to the mCRL2 toolkit. These tools are named *pbesparelm* and *pbesconstelm*. As the names suggest, *pbesparelm* detects and removes superfluous parameters from PBESs and *pbesconstem* detects if parameters are constant and replaces these parameters by its constant value.

## Future work

In [GW05] a technique is proposed that detects patterns in a PBES, for which the solution is known. Using these patterns could have a positive effect on the computation time needed to

solve a PBES. It may be fruitful to research this topic further.

The algorithm *ConstElm* detects constant values in a PBES. That a parameter is a constant is an invariant. As stated in [OW08], the (automatic) detection of invariants can improve the power of e.g. symbolic model checking. This makes detection of invariants an interesting topic for further research.

As a final note, the mCRL2 toolkit would benefit from a working implementation of a PBES solver that uses symbolic approximation. This because symbolic approximation can solve a wide range of PBESs, which cannot be solved using instantiation.

# Appendix A

### Implementation of ParElm

Here we present the functions needed for the algorithm $ParElm$. The first two functions we need are $DPf$ and $MakeEdges$.

---

$DPf(\varphi)$

---

Input:
$\varphi$ : A predicate formula
Output:
The set of influential data terms in predicate formula $\varphi$

---

if $\varphi = \varphi_1 \wedge \varphi_2$ $\rightarrow$ $return\ DPf(\varphi_1) \cup DPf(\varphi_2)$
[] $\varphi = \varphi_1 \vee \varphi_2$ $\rightarrow$ $return\ DPf(\varphi_1) \cup DPf(\varphi_2)$
[] $\varphi = \forall_{e:E}.\varphi_1$ $\rightarrow$ $return\ DPf(\varphi_1) \backslash \{e\}$
[] $\varphi = \exists_{e:E}.\varphi_1$ $\rightarrow$ $return\ DPf(\varphi_1) \backslash \{e\}$
[] $\varphi = b$ $\rightarrow$ $return\ FV(b)$
[] $\varphi = X_j(\overrightarrow{e})$ $\rightarrow$ $return\ \emptyset$
fi

---

---

$MakeEdges(\varphi, \overrightarrow{d_i})$

---

Input:

$\varphi$ : a predicate formula, initially $\varphi_i$ of an equation $X_i$

$\overrightarrow{d_i}$ : The set of parameters of $X_i$

Output:

The set of edges for the dependency graph, obtained from $\varphi_i$ and $\overrightarrow{d_i}$

---

if $\varphi = \varphi_1 \wedge \varphi_2$ $\rightarrow$ $return\ MakeEdges(\varphi_1, \overrightarrow{d_i}) \cup MakeEdges(\varphi_2, \overrightarrow{d_i})$

[] $\varphi = \varphi_1 \vee \varphi_2$ $\rightarrow$ $return\ MakeEdges(\varphi_1, \overrightarrow{d_i}) \cup MakeEdges(\varphi_2, \overrightarrow{d_i})$

[] $\varphi = \forall_{e:E}.\varphi_1$ $\rightarrow$ $return\ MakeEdges(\varphi_1, \overrightarrow{d_i})$

[] $\varphi = \exists_{e:E}.\varphi_1$ $\rightarrow$ $return\ MakeEdges(\varphi_1, \overrightarrow{d_i})$

[] $\varphi = b$ $\rightarrow$ $return\ \emptyset$

[] $\varphi = X_j(\overrightarrow{e})$ $\rightarrow return\ \{((\overrightarrow{d_i})_a, (\overrightarrow{d_j})_b) | (\overrightarrow{d_i})_a \in FV((\overrightarrow{e})_b)\}$

fi

---

The algorithm $DPf$ returns a set of influential parameters in data terms of predicate formula $\varphi$ ($DP_f(\varphi)$), and $MakeEdges$ returns edges of the dependency graph for an equation $\sigma X_i(\overrightarrow{d_i} : \overrightarrow{D_i}) = \varphi_i$. Not the whole equation is the input of the function $MakeEdges$, only the necessary parts, which are the predicate formula $\varphi_i$ and the parameter list $\overrightarrow{d_i}$. $MakeEges$ searches the predicate formula $\varphi_i$ for predicate variables. If a predicate variable $X_j(\overrightarrow{e})$ is encountered, edges are added from $(\overrightarrow{d_i})_a$ to $(\overrightarrow{d_j})_b$ *if* parameter $(\overrightarrow{d_i})_a \in FV((\overrightarrow{e})_b)$. Note the similarity of the two functions $DPf$ and $MakeEdges$. They can be calculated in one run by tupling the set of influential parameters and the set of edges.

Using $MakeEdges$ that makes edges of one equation, we can now introduce the function $MakeGraph$, that makes the dependency graph of a whole PBES.

---

$MakeGraph(\mathcal{E})$

---

Input:

$\mathcal{E}$ : A PBES

Output:

The dependency graph G = (V,E) of PBES $\mathcal{E}$

---

if $\mathcal{E} = \epsilon$ $\rightarrow$ $return\ (\emptyset, \emptyset)$

[] $\mathcal{E} = (\sigma X(\overrightarrow{d} : \overrightarrow{D}) = \varphi)\mathcal{E}'$ $\rightarrow$ $return\ (\overrightarrow{d}, MakeEdges(\varphi, \overrightarrow{d})) \bigcup MakeGraph(\mathcal{E}')$

fi

---

$MakeGraph$ takes as argument a PBES. The result of this function is a graph $(V, E)$, where $V$ are the vertices and $E$ are the edges of the graph. We introduce a binary operator $\bigcup$ which

works on tuples of sets and has the following property:
$(V_1, E_1) \bigcup (V_2, E_2) = (V_1 \cup V_2, E_1 \cup E_2)$

Using $DPf$, which returns a set of influential parameters in data terms of a predicate formula, we introduce $DPp$.

---

$DPp(\mathcal{E})$

---

Input:
$\mathcal{E}$ : A PBES
Output:
The set of all influential data terms in all predicate formulas

---

if $\mathcal{E} = \epsilon$ $\rightarrow$ $return\ \emptyset$
[] $\mathcal{E} = (\sigma X(\overrightarrow{d} : \overrightarrow{D}) = \varphi)\mathcal{E}'$ $\rightarrow$ $return\ DPf(\varphi)\ \cup\ DPp(\mathcal{E}')$
fi

---

$DPp$ returns the set $DP_p(\mathcal{E})$ of all influential parameters that occur in data terms of a whole PBES. The algorithms $MakeGraph$ and $DPp$ can be combined by tupling the return values. When these two algorithms are combined it can work with the combined version of $DPf$ and $MakeGraph$.
With the functions declared above we can introduce the function $ParElm$, which marks all influential parameters.

---

$ParElm(\mathcal{E})$

---

Input:
$\mathcal{E}$ : A PBES
Output:
The set of all influential parameters of a PBES

---

$(V, E) = MakeGraph(\mathcal{E})$
$Infl := DPp(\mathcal{E})$;
$S := Infl$;
while $S \neq \emptyset$ do
    $n$ :=an element of S;
    $S := S \backslash \{n\}$;
    forall $(n', n) \in E$ do
        if $n' \notin Infl \ \rightarrow \ S := S \cup \{n'\}$ fi;
        $Infl := Infl \cup \{n'\}$;
    od
od  return $Infl$

---

The function $ParElm$ makes a graph using $MakeGraph$ and does a reachability test. This is done by marking all vertices that have a directed edge to an influential parameter as influential. The result is a marking of all influential parameters $\mathcal{I}(\mathcal{E})$. All parameters that are not marked are superfluous and can be removed.

# Appendix B

### Implementation of ConstElm

In this Section we present an implementation of the Algorithm *ConstElm*.

The functions $TC$ and $FC$ below uses the rules of Definition 4.1.6 to return a *true-condition* and a *false-condition* of a node.

---

fun TC( $\varphi$ )

---

Input: $\varphi$: A predicate formula
Output: The *true-condition* of the root of $\varphi$

---

| | |
|---|---|
| if $(\varphi = b)$ | $\rightarrow$ *return* $b$ |
| [] $(\varphi = X)$ | $\rightarrow$ *return* $false$ |
| [] $(\varphi = \varphi_1 \wedge \varphi_2)$ | $\rightarrow$ *return* $TC(\varphi_1) \wedge TC(\varphi_2)$ |
| [] $(\varphi = \varphi_1 \vee \varphi_2)$ | $\rightarrow$ *return* $TC(\varphi_1) \vee TC(\varphi_2)$ |
| [] $(\varphi = \forall_{d:D}.\varphi_1)$ | $\rightarrow$ *return* $\forall_{d:D}.TC(\varphi_1)$ |
| [] $(\varphi = \exists_{d:D}.\varphi_1)$ | $\rightarrow$ *return* $\exists_{d:D}.TC(\varphi_1)$ |
| fi | |

---

fun FC( $\varphi$ )

Input: $\varphi$: A predicate formula
Output: The *false-condition* of the root of $\varphi$

| | |
|---|---|
| if $(\varphi = b)$ | $\rightarrow return\ \neg b$ |
| [] $(\varphi = X)$ | $\rightarrow return\ false$ |
| [] $(\varphi = \varphi_1 \wedge \varphi_2)$ | $\rightarrow return\ FC(\varphi_1) \vee FC(\varphi_2)$ |
| [] $(\varphi = \varphi_1 \vee \varphi_2)$ | $\rightarrow return\ FC(\varphi_1) \wedge FC(\varphi_2)$ |
| [] $(\varphi = \forall_{d:D}.\varphi_1)$ | $\rightarrow return\ \exists_{d:D}.FC(\varphi_1)$ |
| [] $(\varphi = \exists_{d:D}.\varphi_1)$ | $\rightarrow return\ \forall_{d:D}.FC(\varphi_1)$ |
| fi | |

Before we can apply the rules of Definition 4.1.10 we first introduce some help functions:

fun AddQ($Q_{d:D}$, $cs$ )

Input:
$Q_{d:D}$: A quantification
$cs$: A set of conditions
Output:
The quantifcation $Q_{d:D}$ applied to all elements of the set $cs$

$return\ \{Q_{d:D}.c | c \in cs\}$

Definition 4.1.10 only collects true/false conditions for a single predicate variable, here we introduce a functions that collects the conditions for all predicate variables of a predicate formula. To do this we need a *bag* $B$ of predicate variables, each predicate variable of the bag contains a set of *true-conditions* and a set of *false-conditions*. We denote the set of *true-conditions* and *false-conditions* for predicate variable $X(\overrightarrow{e})$ as $X(\overrightarrow{e})^{tcs}_{fcs}$. To add a true/false conditions to all predicate variables of bag $B$ we introduce the function $AddC(B, tc, fc)$

---

fun AddC(B,tc,fc)

---

Input:

$B$: A bag of predicate variables with a set of true conditions and a set of false conditions

$tc$: A *true-condition* that must be added to the set of true conditions of each predicate variable of $B$

$fc$: A *false-condition* that must be added to the set of false conditions of each predicate variable of $B$

Output:

The bag $B$ with the true/false condition added to set of true/false conditions of predicate variables of $B$

---

$return\ \{X^{tcs \cup tc}_{fcs \cup fc} | X^{tcs}_{fcs} \in B\}$

---

With the functions *AddQ* and *AddC* we can now make a bag of predicate variables, each with a set of *true-condition* and a set of *false-condition* which are encountered on the path from root to that variable.

---

fun MakeBag($\varphi$)

---

Input:

$\varphi$: A predicate formula for which the bag is made

Output:

A bag $B$ with contains predicate variables, each predicate variable $X$ has a set of *true-conditions* and a set of *false-conditions* which are on the path from the root to $X$.

---

if $(\varphi = b)$ $\rightarrow$ $return\ \emptyset$

[] $(\varphi = X)$ $\rightarrow$ $return\ \{X^{false}_{false}\}$

[] $(\varphi = \varphi_1 \oplus \varphi_2)$ $\rightarrow$ $bg := MakeBag(\varphi_1) \cup MakeBag(\varphi_2)$;
$bg := AddC(bg, TC(\varphi), FC(\varphi))$;
$return\ bg$

[] $(\varphi = \forall_{d:D}.\varphi_1)$ $\rightarrow$ $bg := MakeBag(\varphi_1)$;
$bg := \{X^{AddQ(\forall_{d:D}, tcs)}_{AddQ(\exists_{d:D}, fcs)} | X \in bg\}$;
$bg := AddC(bg, TC(\varphi), FC(\varphi))$;
$return\ bg$

[] $(\varphi = \exists_{d:D}.\varphi_1)$ $\rightarrow$ $bg := MakeBag(\varphi_1)$;
$bg := \{X^{AddQ(\exists_{d:D}, tcs)}_{AddQ(\forall_{d:D}, fcs)} | X \in bg\}$;
$bg := AddC(bg, TC(\varphi), FC(\varphi))$;
$return\ bg$

fi

---

Note that the functions $TC$, $FC$ and $MakeBag$ are all depth first algorithms that can be combined to one algorithm which only needs to parse a predicate formula once. The function $MakeBag$ creates a bag of predicate variables with corresponding *true-conditions* and *false-conditions*. Using this bag we can make a dependency graph. We first introduce a helper

function $Neg$ which works as defined in Definition 4.1.11.

---

fun Neg(cs)

---

Input:
$cs$: A set of conditions
Output:
The conjunction of the negation of all elements of the set $cs$

---

$return\ \{\bigwedge \neg c | c \in cs\}$

---

We now have all ingredients to construct a dependency graph with conditions on the edges.

---

fun MakeGraph($\mathcal{E}$)

---

Input:
$\mathcal{E}$: A PBES
Output:
The dependency graph for $\mathcal{E}$

---

$V := \emptyset;$
$E := \emptyset;$
$forall\ \sigma X_i(\overrightarrow{d_i} : \overrightarrow{D_i}) = \varphi_i\ \ do$
$\quad V := V \cup X_i;$
$\quad bg := MakeBag(\varphi_i);$
$\quad forall\ X_j(\overrightarrow{e})^{tcs}_{fcs} \in bg\ \ do$
$\quad\quad cond := Neg(tcs) \wedge Neg(fcs);$
$\quad\quad E := E \cup \{X_i, \overrightarrow{d_j} := \overrightarrow{e}, cond, X_j\};$
$\quad od$
$od$
$; return\ (V, E)$

---

We can now give an algorithm for $ConstElm$ that marks parameters that can be replaced by constants.

fun ConstElm($\mathcal{E}$, $\kappa$)

(V,E) := $MakeGraph(\mathcal{E})$;
$A$ := assertions from $\kappa$
$S$ := initial equations from $\kappa$;
do $S \neq \emptyset \rightarrow$
  $v$ :=an element of ($S$);
  $S := S \backslash \{v\}$;
  forall edges $(v, \overrightarrow{d_j} := \overrightarrow{e}, cond, X_j)$ do
    if $cond$ does not reduce to $false \rightarrow$
      $forall\ 1 \leq i \leq len(\overrightarrow{e})$ *do*
        $r := reduce\ (\overrightarrow{e})_i$;
        $if\ r = NaC \wedge \{(\overrightarrow{d_j})_i = NaC\} \notin A \wedge \{(\overrightarrow{d_j})_i = c_1\} \notin A$ $\rightarrow A := A \cup \{(\overrightarrow{d_j})_i = NaC\}$; $S := S \cup X_j$
        [] $r = NaC \wedge \{(\overrightarrow{d_j})_i = c_1\} \in A$ $\rightarrow A := A \setminus \{(\overrightarrow{d_j})_i = c_1\}$; $A := A \cup \{(\overrightarrow{d_j})_i = NaC\}$; $S := S \cup X_j$
        [] $r = c_1 \wedge \{(\overrightarrow{d_j})_i = c_2\} \in A \wedge c_1 \neq c_2$ $\rightarrow A := A \setminus \{(\overrightarrow{d_j})_i = c_1\}$; $A := A \cup \{(\overrightarrow{d_j})_i = NaC\}$; $S := S \cup X_j$
        [] $r = c_1 \wedge \{(\overrightarrow{d_j})_i = c_2\} \notin A \wedge \{(\overrightarrow{d_j})_i = NaC\} \notin A$ $\rightarrow A := A \cup \{(\overrightarrow{d_j})_i = c_1\}$; $S := S \cup X_j$
        [] *otherwise* $\rightarrow$ *skip*
        fi
    od
  fi
 od
od

This algorithm takes as argument a PBES $\mathcal{E}$ and a predicate $\kappa$. First the initial state is created by adding assertions to the set of assertions $A$, and by adding vertices to the set of vertices $S$. This information is gathered from the predicate $\kappa$. The algorithm than takes a vertex $v$ from $S$ and removes it from the set, afterward all outgoing edges, ine. starting from $v$, are examined. If the condition on the edge can be reduced to false, the edge is skipped. The method used to reduce the condition is left to the implementor, note however that the condition is a predicate and the value of some variables may not be known. Because of this the condition cannot always be reduced to *true* or $false$. If the condition does not reduce to $false$, the edge is examined; for each parameter the value is calculated and put in variable $r$. Here again, how the value is calculated is left to the implementor. Now a case distinction is made, which follow the algorithm described in Section 4.1.

# Appendix C

### mCRL2 specification of the alternating bit protocol

```
sort
  D     = Nat;
  Error = struct e;
act
  r1,s4: D;
  s2,r2,c2: D # Bool;
  s3,r3,c3: D # Bool;
  s3,r3,c3: Error;
  s5,r5,c5: Bool;
  s6,r6,c6: Bool;
  s6,r6,c6: Error;
  i;
proc
  S(b:Bool)     = sum d:D. r1(d).T(d,b);
  T(d:D,b:Bool) = s2(d,b).(r6(b).S(!b)+(r6(!b)+r6(e)).T(d,b));

  R(b:Bool)     = sum d:D. r3(d,b).s4(d).s5(b).R(!b)+
                  (sum d:D.r3(d,!b)+r3(e)).s5(!b).R(b);

  K             = sum d:D,b:Bool. r2(d,b).(i.s3(d,b)+i.s3(e)).K;

  L             = sum b:Bool. r5(b).(i.s6(b)+i.s6(e)).L;
init
  allow({r1,s4,c2,c3,c5,c6,i},
    comm({r2|s2->c2, r3|s3->c3, r5|s5->c5, r6|s6->c6},
        S(true) || K || L || R(true)
    )
  );
```

# Bibliography


[CLRS01] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[DPWar] A. van Dam, B. Ploeger, and T.A.C. Willemse. Instantiation for parameterised boolean equation systems. CS-Report 08-11, Eindhoven University of Technology, Eindhoven University of Technology, Department of Computer Science, 2008, to appear.

[GL02] J.F. Groote and B. Lisser. Computer assisted manipulation of algebraic process specifications. *SIGPLAN Not.*, 37(12):98–107, 2002.

[GMR+07] J.F. Groote, A. Mathijssen, M.A. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A.H. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*, number 06351 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[GR01] J.F. Groote and M.A. Reniers. Algebraic process verification. In *Handbook of Process Algebra, chapter 17*, pages 1151–1208, 2001.

[GW04] J.F. Groote and T.A.C. Willemse. A checker for modal formulae for processes with data. *FMCO 2003*, pages 223 – 239, 2004.

[GW05] J.F. Groote and T.A.C. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, pages 332–369, 2005.

[Mad97] A.H. Mader. Verification of modal properties using boolean equation systems. 1997.

[OW08] S.M. Orzan and T.A.C. Willemse. Invariants for parameterised boolean equation systems. *CONCUR'08*, 2008.

[SW89] C. Stirling and D. Walker. *Local Model Checking in the Modal Mu-Calculus*. Springer Berlin / Heidelberg, 1989.