TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

Instantiation of Parameterised Boolean Equation Systems

by

A. van Dam

SUPERVISORS

Dr. Ir. T.A.C. Willemse
Prof. Dr. Ir. J.F. Groote

August 2007, Eindhoven

Abstract

In the field of model checking, several techniques are researched on verifying properties on systems. Some of those techniques are used in the *mCRL2-toolset*. Using the *mCRL2-toolset* it is possible to specify and analyse systems. In this thesis we will discuss research which is conducted on techniques to instantiate parameterised boolean equation systems (PBESs), which are part of the *mCRL2-toolset*.

PBESs are used to represent a specification of a system combined with a given property one wants to check on that system. Using symbolic approximation techniques they can be solved in some cases, but using this technique will often not lead to a solution for the PBES. Because PBESs use data in their specification, it is possible to instantiate this data in PBESs. We have developed two techniques to instantiate PBESs.

The first technique, called the finite approach, reduces the complexity of a PBES by instantiating all finite data types of the predicate variables. On the resulting PBES symbolic approximation techniques can be applied, which can lead to a solution while the original PBES could not be solved using those techniques.

The second technique is the lazy approach, which assumes that the PBES has to be solved in the initial state. It takes the initial state of a PBES and only computes those equations needed to solve the PBES in the initial state. This approach leads to a boolean equation system (BES), which can be solved using BES-solving techniques like gauß elimination.

We have conducted case studies on the alternating bit protocol and Lamport's bakery protocol, which shows the differences between the two approaches.

Contents

Chapter 1

Introduction

Model checking is a technique for verifying properties of concurrent systems (systems in which processes can communicate with each other). The properties are often described by means of an expression in a *temporal logic.* The system itself is represented using a specification language like mCRL2. The verification concludes with a positive result if the property holds on the system and a negative result (with in some cases an error trace) if the property does not hold on the system. Ideally the verification process is done automatically, but a lot of verifications are not yet possible with the existing tools, for example because the state space of systems can be of infinite size.

In the last decade the research on model checking has led to several techniques for model checking. These techniques are techniques to reduce state spaces of systems or represent those state spaces more compact. Some examples of such model checking techniques are:
- Symbolic model checking; e.g. the use of symbolic approximation techniques [GrW05a] and binary decision diagrams (BDDs, [McM92]).
- Partial order reduction; reduces systems by removing paths to states, if those states can be reached by executing the same steps in a different order.
- Abstraction; simplifying the state space of a system by removing those parts of that system which are not of interest for the property which is verified.

Several toolsets have been developed (and are still under development) to make verifying properties on systems using model checking easier. Examples of those toolsets are the *mCRL2-toolset* [MCRL2], developed at the Eindhoven University of Technology (TU/e) and the *CADP-toolset* [CADP], developed at the French National Institute for Research in Computer Science and Control (INRIA). The research in this thesis is based on the *mCRL2-toolset.*

The *mCRL2-toolset* uses the mCRL2 specification language, which is a process algebra with data and timing. With this language concurrent systems can be specified using a data specification and a process specification. Using the tools of the toolset, a specification of a system can be transformed to a basic form called *linear process specification* (LPS). An LPS can be manipulated to simplify the LPS, a *labelled transition system* (LTS) can be generated out of that LPS and the LPS is the basis of a translation to a *parameterised boolean equation system* (PBES).

The state space of an LPS is often of infinite size. To verify properties on that infinite state space, PBESs [GrW05] are used, which are the result of combining a specification of a system (represented by an LPS) with a property (represented by a formula in the first order modal μ-calculus [GrW05a]). PBESs are therefore an important part of model checking in the *mCRL2-toolset.* For instance, answering the question if a property holds on the system, can be done by computing the solution of a PBES.

Techniques to solve subsets of PBESs are available. However, there is a need for techniques to solve more PBESs. Instantiation of PBESs is a step towards solving a subset of PBESs. By instantiating PBESs it is possible to reduce the complexity of PBESs, eventually leading to boolean equation systems (BESs, [Mad97]), which can be solved using BES-solvers. In this thesis we discuss techniques to instantiate PBESs. The discussed techniques are implemented in the tool *pbes2bes*.

1.1 Related work

The *μCRL-toolset* [MCRL], the predecessor of the *mCRL2-toolset*, contains a tool *mucheck* which can solve a subset of PBESs using symbolic approximation techniques [GrW05a]. To use symbolic approximation techniques in the *mCRL2-toolset* a tool *pbesssolve*, based on the tool *mucheck*, is currently implemented. Using the techniques defined in this thesis, it is possible to increase the performance of symbolic approximation techniques, because the PBESs which have to be solved are reduced in complexity.

In the *CADP-toolset* [CADP], there are tools available which create a BES out of an LTS and a property in terms of a temporal logic formula. Because an LTS is needed to create a BES, it is not possible to consider systems with in infinite state space. The toolset also contains a BES-solver which can solve *alternation-free BESs* [Mat03]. Because only alternation-free BESs can be solved, the complexity of the temporal formulae which can be checked are restricted. Also, a lot of BESs which are created using the techniques we define in this thesis will not be solvable with the BES-solver in the *CADP-toolset*.

The concurrency workbench [CPS93] supports several verification methods, including verification using tableau-based methods on a modal logic based on the propositional (modal) μ-calculus [Koz83]. Specifications of systems are written in this logic and the workbench can automatically check those specifications.

1.2 Overview of this thesis

As data is an important of PBESs, we discuss the way data is used in PBESs in Chapter 2.
The tools are implemented for the *mCRL2-toolset*. Chapter 3 gives an overview of the toolset.
To be able to instantiate PBESs, we need to define what PBESs are. This is done in Chapter 4.
In Chapter 5, we define two instantiation techniques. The first technique is the finite approach, which instantiates all finite data types in the PBES. This approach will always terminate, because infinite data types are not dealt with. The result of the finite approach is a PBES. The finite approach is discussed in Section 5.2. The second approach is the lazy approach, which assumes a PBES has to be solved in the initial state. Therefore the result will be a BES with only those equations the initial state depends on. The lazy approach is discussed in Section 5.3
We implemented those techniques in the tool *pbes2bes*. We discuss issues which arose during the implementation in Section 5.4. Examples of those issues are the existence of free data variables (data variables for which it is not important what value is instantiated for that data variable) in PBESs and the fact that no rewriter for predicate formulae was available.
In Section 5.5 we describe a number of possible optimisations, which can increase the performance of the tool and further decrease the complexity of the resulting PBESs and BESs.
We conducted case studies on a number of properties on the alternating bit protocol and Lamport's bakery protocol, which we will discuss in Chapter 6.
In Chapter 7, we state a number of conclusions on the research and we identify future work.

Chapter 2

Data

Data is an important part of many systems. Any system which must be able to exchange information uses data. An example of such systems are communication protocols like the alternating bit protocol [BaW90]. To be able to describe systems which use data, in the mCRL2 specification language, systems can be specified with data, which makes it possible to verify data-dependent systems. PBESs, which are part of the *mCRL2-toolset*, also use data in their definitions. In this chapter we give a short overview of the concepts of data which are used in this thesis.

Data is treated in an abstract way. We assume that there are non-empty *data sorts*, which are generally written using letters D, E, F, but also includes the sort $\mathbb{B}$, which is the sort over the boolean values, containing $\top$ and $\bot$, which represent true and false respectively.

Data sorts consist of one or more *data constructors*, which are the basic elements of a data sort. A data sort is called *finite* if all the data constructors of that data sort are finite. A data constructor is finite if it is a constant with the same type as the data sort it defines (like $\bot$ and $\top$ for booleans), or a function containing only finite data sorts in its domain.
A data sort is called *countable* (or denumerable) if there is a mapping from the data sort to the natural numbers (e.g. the elements in the set can be counted). Examples of countable data sorts are $\mathbb{N}$ (natural numbers) and $\mathbb{Z}$ (integers). An example of a data sort which is not countable is $\mathbb{R}$ (real numbers).

A *data term* is an element which can be built from the constructors of and functions over a specific data sort. We assume that there is a set $\mathcal{D}$ of data variables, with elements $d, d_1, \ldots$ and a data language that is sufficiently rich to denote all relevant data terms, for instance $3 + d_1 \geq d_2$. If a data term is of a form where only the data constructors are used to represent that data term, we will call this a *data element* or value in some cases.

A *closed data term* is a data term which can be rewritten to a data element. For a closed data term e, we assume there is an interpretation function $[\![e]\!]$, that maps e to the data element it represents.

An *open data term* is a data term which can not be rewritten to a data element. An example term is the data term $n+1$ of data sort $\mathbb{N}$, where n is a data variable of the data sort $\mathbb{N}$. For open data terms a **data environment** $\varepsilon : D \rightarrow E$ is needed, that maps each variable from D to a data value (a specific data term of a data sort) of the correct data sort (E). The interpretation of an open data term e of data sort $\mathbb{B}$, denoted as $[\![e]\!]\varepsilon$ is given by $[\![\varepsilon(e)]\!]$, where ε is extended to data terms in the standard way.

Chapter 3

mCRL2

The tools which are written during the research described in this thesis, are part of the *mCRL2-toolset* [MCRL2]. This toolset is the successor of the *μCRL-toolset* [GrP94, GrR01]. With the *mCRL2-toolset* it is possible to specify systems with data and timing using the mCRL2 specification language [GMP06, GMR07], which is based on the Algebra of Communicating Processes (ACP) [BaW90]. Using the toolset it is possible to transform the specification into other conceptual formats as well as manipulate the specification in several ways. An overview of the toolset is given in figure 3.1.

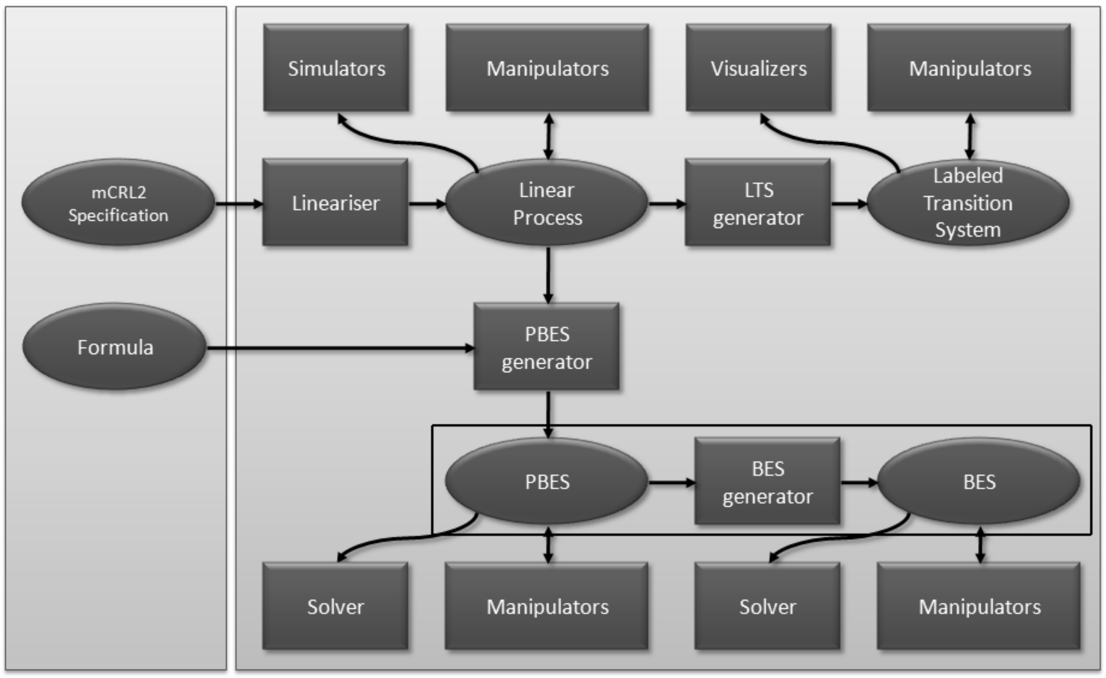

Figure 3.1: Overview of the mCRL2 toolset

This thesis is about the implementation of the BES-generator, using PBESs and BESs, as we have marked in the above figure.

3.1 Concepts

In this section we will briefly describe the concepts of the *mCRL2-toolset.*

Specification

The specification describes the behavior of a system in terms of a data specification and a process specification. In this thesis only specifications without time are considered.

Data Specification Data is an important aspect of many systems. Every system capable of exchanging information with its environment uses data. Abstracting from the data is often undesirable. Therefore in mCRL2 a system can be specified including the data it depends on. The data specification is based on higher-order abstract data types.

A new data type can be declared using the keyword **sort**. To specify the domain of the sort, constructors (**cons**) are used. Functions over the sorts are declared using **map**. These maps are defined using equations (**eqn**), which can use variables (**var**), denoting arbitrary elements of the sort.

Example 3.1. The following data specification specifies a sort D, with constructors $d1$ and $d2$ and a map inverse. The inverse of $d1$ is $d2$ and vice versa.

sort	$D;$
cons	$d1, d2 : D$
map	$inverse : D \to D;$
var	$d : D;$
eqn	$inverse(d1) = d2;$
	$inverse(d2) = d1;$
	$inverse(inverse(d)) = d;$

Besides the user defined data types, there are a number of concrete data types. These concrete data types consist of *standard data types and functions* and *type constructors.* The standard data types contain positive ($\mathbb{N}^+$), natural ($\mathbb{N}$), integer ($\mathbb{Z}$) and real ($\mathbb{R}$) numbers, with a number of relations and operators on these numbers, such as: $<$, $\leq$, $\geq$, $>$, -, +, *, **div** and **mod**. Also, there is a data type which represents the booleans ($\mathbb{B}$), with the constants *true* and $false$, and operators $\wedge$, $\vee$, $\Rightarrow$ and $\neg$. For all data types, operators denoting equality and inequality are available as well as an if-then-else function.
Type constructors are used in the declaration of lists, sets and bags containing elements of any sort (both user defined or predefined). A list of booleans for instance, is defined as List(Bool) and has constructors $[] : List(Bool)$ (representing the empty list) and $\rhd$: $Bool \to List(Bool) \to List(Bool)$ (representing a non-empty list). An example of a predefined function over lists is the concatenation (++), which takes two lists and puts all elements of the second list recursively at the end of the first list.

Process specification The process specification defines the communicating processes of the system and an initial state. The behavior of each process is defined by recursive definitions. The basic elements of such definitions are actions (representing an atomic event) and deadlock (denoted by δ). These basic elements can be composed to process expressions by *sequential composition* and *alternative composition.* When using sequential composition of p and q (denoted p.q), the process first executes p and when p is finished it executes q. Alternative composition of p and q (denoted p+q), chooses non-deterministically between execution of either p or q.

Example 3.2. Extending example 3.1 we can add a process P, with data parameter d of sort D, which executes an action a followed by P with the inverse of d or it executes an action b, followed by P with d. As initial state we choose $P(d1)$:

act a, b;
proc $P(d{:}D) = a.P(inverse(d)) + b.P(d)$;
init $P(d1)$;

Processes can be combined using *parallel composition* ($\|$) and *left merge* ($\lfloor\!\lfloor$). Parallel composition $p\|q$ is the interleaving and synchronisation of the actions of p with those of q. The left merge $p\lfloor\!\lfloor q$ executes the first action of p, followed by the parallel composition of the remainder of p and q. It is also possible to define *multi-actions* ($|$), which is a bag of actions which are executed at the same time; $a|b$ executes actions a and b at the same time.

Linear Process Specification

Because a general specification is not suitable for manipulation, and a transition system generated out of that specification can be of infinite size, an mCRL2-specification is transformed to a *linear process specification* (LPS). The behavior of the system in the mCRL2-specification is preserved in the LPS.
In the data specification of an LPS both the user defined data, as well as the concrete data types needed for the process specification are present. The process specification in the LPS is of a form which is usable for further manipulation. These manipulators can simplify an LPS in some cases.

Example 3.3. Using the previous example of the data specification 3.1 and process specification 3.2, an LPS can be generated. This LPS contains all definitions for all sorts, constructors, maps and equations needed for the process specification. The linear process specification is as follows:

act a, b;
proc $P(d{:}D) = true \rightarrow a.P(inverse(d))$
$\quad + \; true \rightarrow b.P(d)$;
init $P(d1)$;

Labelled Transition System

From an LPS a *labelled transition system* (LTS) can be generated. The LTS contains the behavior of the model as a statespace. With visualisers such as *ltsview* or *ltsgraph* the LTS can be visualised.

μ-Calculus formula

A μ-calculus formula represents a specific property of a system. The grammar of a first order μ-calculus formula φ is the following:

$$\begin{aligned}\varphi \; ::= \; & c \mid true \mid false \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \forall d{:}D.\varphi \mid \exists d{:}D.\varphi \mid \\ & \langle\alpha\rangle\varphi \mid [\alpha]\varphi \mid X(e) \mid (\mu X(d{:}D).\varphi)(e) \mid (\nu X(d{:}D).\varphi)(e)\end{aligned}$$

Where c is a boolean expression, e is an arbitrary data term, D is a possibly empty set of data types and α is an arbitrary action formula which is defined by the following grammar:

$$\alpha \; ::= \; a(e) \mid \top \mid \bot \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \exists d{:}D.\alpha \mid \forall d{:}D.\alpha$$

The semantics of the first order μ-calculus and action fomulae is described in [GrW05a].
A typical example of a μ-calculus formula is $\nu X.\langle true\rangle true \wedge [true]X$, which represents the property that a system is deadlock free (after an arbitrary number of steps, there is always another step possible).

Parameterised Boolean Equation System

A PBES is generated when an LPS and a μ-calculus formula are combined. In Chapter 4, PBESs are more thoroughly described.

Example 3.4. The resulting PBES after combining the LPS of example 3.3 and the property which describes the absence of deadlocks in the system is the following:
$\nu\ X(d{:}D) = (\ \neg\ true \vee X(inverse(d)\ ) \wedge (\ \neg\ true \vee X(d)\ ) \wedge (true \vee true)$, which can be rewritten to:
$\nu\ X(d{:}D) = X(inverse(d)) \wedge X(d)$

Chapter 4

Parameterised boolean equation systems

PBESs [GrW05] or *First order BESs* are a special case of fixpoint equation systems [Mad97] and first appeared in [GrM99]. A PBES is a sequence of fixpoint equations (which we will call parameterised boolean equation or *equation* in this thesis), where each fixpoint equation has the form $\sigma X(d_1{:}D_1, \ldots, d_n{:}D_n) = \varphi$, where σ is either μ denoting a minimal fixpoint or ν denoting a maximal fixpoint.

Each parameterised boolean equation has at its left hand side a *predicate variable* $X \in \mathcal{X}$ (where $\mathcal{X}$ is the set of all predicate variables) which binds zero or more data variables $d_1, \ldots, d_n$. The signature of X is $D_1 \times \ldots \times D_n \to \mathbb{B}$. For simplicity, the sequence of data variables will often be denoted as $d{:}D$ in the theoretical considerations. A predicate variable with an instantiation for the sorts of the signature of that predicate variable is called a *predicate variable instantiation.* For example, $X(3)$ is a predicate variable instantiation of predicate variable $X(n{:}\mathbb{N})$.

The right hand side of a parameterised boolean equation is a *predicate formula*, denoted by φ. It contains data terms, boolean connectives, predicate variables and quantifiers over (possible infinite) data domains and data. Such predicate formulae are defined by the following grammar:

$$\varphi ::= \ b \mid \top \mid \ \bot \ \mid X(e) \mid \varphi \ \wedge \ \varphi \mid \varphi \ \vee \ \varphi \mid \forall \ d{:}D.\varphi \mid \exists \ d{:}D.\varphi$$

Where b is a data term of data sort $\mathbb{B}$, $\top$ and $\bot$ are elements of data sort $\mathbb{B}$, X is a predicate variable, e is a data term and d is a data variable of sort D.

Predicate formulae are interpreted in a context of a data environment ε and a predicate environment $\eta : \mathcal{X} \to (D \to \mathbb{B})$. For an arbitrary environment θ (denoting ε or η), we write $\theta[d := v]$ for the environment θ in which variable d is assigned the value v and all other variables remain unchanged. For a predicate formula φ, a data environment ε and a predicate environment η, $\varphi(\eta\varepsilon)$ denotes the formula φ where all free predicate variables X are valued $\eta(X)$ and all free data variables d are valued $\varepsilon(d)$.

Definition 4.1. Semantics of predicate formulae
Let ε be a data environment and η be a predicate environment. The interpretation $[\![\varphi]\!]\eta\varepsilon$ maps a predicate formula φ to *true* or *false*, and is inductively defined as follows:

$$\begin{array}{rcl}
[\![\top]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & true \\
[\![\bot]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & false \\
[\![b]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & [\![b]\!]\varepsilon \\
[\![X(e)]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & \eta(X)([\![e]\!]\varepsilon) \\
[\![\varphi_1 \wedge \varphi_2]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & [\![\varphi_1]\!]\eta\varepsilon \ and \ [\![\varphi_2]\!]\eta\varepsilon \\
[\![\varphi_1 \vee \varphi_2]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & [\![\varphi_1]\!]\eta\varepsilon \ or \ [\![\varphi_2]\!]\eta\varepsilon \\
[\![\forall d{:}D.\varphi]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & \begin{cases} true & if\ for\ all\ v \in D.[\![\varphi]\!]\eta(\varepsilon[d := v]) \\ false & otherwise \end{cases} \\
[\![\exists d{:}D.\varphi]\!]\eta\varepsilon & \stackrel{\text{def}}{=} & \begin{cases} true & if\ exists\ v \in D.[\![\varphi]\!]\eta(\varepsilon[d := v]) \\ false & otherwise \end{cases}
\end{array}$$

The notion of the right hand side and left hand side as described above, leads to the following definition of a PBES.

Definition 4.2. PBES
A PBES is inductively defined as follows:

- ϵ is an empty PBES.
- If $\mathcal{E}$ is a PBES, then $(\sigma X(d{:}D) = \varphi)\mathcal{E}$ is a PBES.

In the internal format (see appendix A) used in the *mCRL2-toolset*, a PBES is represented as follows:

- Data specification (see section 3.1): Contains all data definitions, which occur in the PBES.
- Equation system: A list of parameterised boolean equations, which represents the equations of the PBES.
- Initial state: To solve a PBES, besides the PBES itself also an initial state is given, which is derived from the initial state of the specification of the system. This initial state is the predicate variable instantiation for which the PBES must be solved. The initial state, denoted by $X(d_{init})$, is one of the predicate variable instantiations in the set of all possible predicate variable instantiations in the PBES: $X(d_{init}) \in \{X(d) | X \in \mathcal{X} \wedge d \in signature(X)\}$, where $signature(X)$ is the set of all possible instantiations for the data variables in X.

The set of predicate variables at the left hand side of the parameterised boolean equations in a PBES $\mathcal{E}$, denoted by $lhs(\mathcal{E})$, is defined as $lhs(\epsilon) \stackrel{\text{def}}{=} \emptyset$ and $lhs((\sigma X(d{:}D) = \varphi)\mathcal{E}) \stackrel{\text{def}}{=} lhs(\mathcal{E}) \cup \{X\}$. All predicate variables which occur at the right hand sides of the parameterised boolean equations in a PBES $\mathcal{E}$ are collected in the set $rhs(\mathcal{E})$. The set of free predicate variables in a PBES $\mathcal{E}$ is defined as follows: $free(\mathcal{E}) = rhs(\mathcal{E}) \backslash lhs(\mathcal{E})$.

In this thesis we only consider closed and well-formed PBESs. A PBES $\mathcal{E}$ is *closed* if all predicate variables which occur in the right hand side can also be found at the left hand side. Thus: $\mathcal{E}$ is closed if $free(\mathcal{E}) = \emptyset$. PBES $\mathcal{E}$ is well-formed if all predicate variables on the left hand side are unique.

To show basic information on PBESs a tool *pbesinfo* is implemented in the *mCRL2-toolset*, which displays if a PBES is well-formed and closed, the number of parameterised boolean equations, the number of μ's, the number of ν's and the predicate variables with their signature. The tool is described further in appendix B.1.

When a PBES is computed with timing, in every right hand side of a parameterised boolean equation a quantifier-expression over the real numbers is present, which in many cases can not be eliminated. Therefore, if a PBES is timed, it will hardly ever be instantiated to a BES, but it will be instantiated to a (possibly simplified) PBES. If real-time aspects are not important in a system, it is possible to create PBESs without timing, so they could be instantiated to BESs. In this thesis we only consider untimed PBESs.

A parameterised boolean equation $\sigma X(d{:}D)=\varphi$ is *solved* if φ does not contain predicate variables. Likewise, a PBES $\mathcal{E}$ is *solved* if all its parameterised boolean equations are solved. A PBES is *solved in* X if the predicate variable X does not occur in any right hand side of $\mathcal{E}$. The solution of a PBES $\mathcal{E}$ in the context of a predicate environment η and a data environment ε is inductively defined as follows, see definition 2.3 of [GrW05]:

Definition 4.3. Solution of a PBES

$$\begin{aligned}
[\epsilon]\eta\varepsilon &\overset{\text{def}}{=} \eta \\
[(\sigma X(d{:}D) = \varphi)\mathcal{E}]\eta\varepsilon &\overset{\text{def}}{=} [\mathcal{E}](\eta[X := \sigma X(d{:}D).\varphi([\mathcal{E}]\eta\varepsilon)])\varepsilon
\end{aligned}$$

Where $\sigma X(d{:}D).\varphi([\mathcal{E}]\eta\varepsilon)$ is defined as:

$$\begin{aligned}
\mu X(d{:}D).\varphi([\mathcal{E}]\eta\varepsilon) &\overset{\text{def}}{=} \bigwedge\{\psi{:}D \to \mathbb{B} \mid \lambda v \in D.[\![\varphi]\!]([\mathcal{E}]\eta[X := \psi]\varepsilon)\varepsilon[d := v] \sqsubseteq \psi\} \\
\nu X(d{:}D).\varphi([\mathcal{E}]\eta\varepsilon) &\overset{\text{def}}{=} \bigvee\{\psi{:}D \to \mathbb{B} \mid \psi \sqsubseteq \lambda v \in D.[\![\varphi]\!]([\mathcal{E}]\eta[X := \psi]\varepsilon)\varepsilon[d := v]\}
\end{aligned}$$

Where $\bigvee$, $\bigwedge$ and $\sqsubseteq$ can be explained as follows. Consider an arbitrary data sort D, and all (total) functions $D \to \mathbb{B}$ over that sort. The set of all such functions is denoted as $[D \to \mathbb{B}]$. The ordering $\sqsubseteq$ on $[D \to \mathbb{B}]$ is defined as $f \sqsubseteq g$ if for all $d{:}D$, $f(d)$ implies $g(d)$. The set $([D \to \mathbb{B}], \sqsubseteq)$ is a complete lattice (see section 2.1 of [Mad97]). For a subset A of $[D \to \mathbb{B}]$, we write $\bigwedge A$ for the *least upper bound*, or *infinum* and $\bigvee A$ for the *greatest lower bound*, or *supremum.*

Example 4.4. Solution of a PBES
Take a PBES $(\nu X = Y)(\mu Y = X)$. For a given predicate environment η, its solution is $\eta[X := \top][Y := \top]$.

Note that if we would have chosen $(\mu Y = X)(\nu X = Y)$, the solution would have been $\eta[X :=\perp][Y :=\perp]$. This shows that the order in which the equations occur is important. Therefore we have to define the order in which parameterised boolean equations occur in a PBES.

Definition 4.5. Order of parameterised boolean equations in a PBES
Let:

- $\mathcal{E}$ be a PBES
- $X(d{:}D)$, $Y(e{:}E)$ denote parameterised boolean equations $\sigma X(d{:}D) = \varphi$, $\sigma' Y(e{:}E) = \psi$
- *position* be a function which maps a parameterised boolean equation $X(d{:}D)$ (a parameterised boolean equation in $\mathcal{E}$) to a position in $\mathcal{E}$, represented by a natural number. This function is defined by:
  position$(X(d{:}D), \mathcal{E}) =$
  $$\begin{cases} 0 & \textit{if } \exists \mathcal{E}'.\ \mathcal{E} = (X(d{:}D))\mathcal{E}' \\ 1 + \textit{position}(X(d{:}D), \mathcal{E}') & \textit{if } \exists \mathcal{E}'. \exists Y(e{:}E).\ \mathcal{E} = (Y(e{:}E))\mathcal{E}' \wedge (Y(e{:}E)) \neq (X(d{:}D)) \end{cases}$$

Then: $X \lhd Y$ if $X \in \textit{lhs}(\mathcal{E})$, $Y \in \textit{lhs}(\mathcal{E})$ and $\textit{position}(\sigma X(d{:}D) = \varphi, \mathcal{E}) < \textit{position}(\sigma Y(e{:}E) = \psi, \mathcal{E})$.

Property 4.6. Properties of $\lhd$.
The order $\lhd$ is irreflexive, asymmetric and transitive:
Let X, Y and Z be predicate variables, for which $X \neq Y$, $Y \neq Z$ and $X \neq Z$ holds. Then:

- Irreflexivity: $\neg(X \lhd X)$
- Asymmetry: $\neg((X \lhd Y) \wedge (Y \lhd X))$
- Transitivity: $((X \lhd Y) \wedge (Y \lhd Z)) \Rightarrow (X \lhd Z)$

Proof. Follows from the irreflexivity, asymmetry and transitivity of $<$ and the definition of *position*. □

4.1 Boolean equation systems

Boolean equation systems are thoroughly described in [Mad97]. BESs are a special case of PBESs, in which there are a number of restrictions:

- The predicate variables must be data-less, and thus are *propositional variables*,
- The predicate formulae of a BES are more restricted and may only consist of $\varphi ::= \top \mid \bot \mid X \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$.

Because BESs are a special case of PBESs, definition 4.5 also applies to BESs.

Example 4.7. To show the differences between PBESs and BESs, we use the PBESs of examples 3.4 and 4.4. The PBES $\nu\ X(d{:}D) = X(\textit{inverse}(d)) \wedge X(d)$ is not a BES, because the predicate variables in the PBES has data variables. On the other hand, the PBES $(\nu X = Y)(\mu Y = X)$ is a BES, because both equations has predicate variables without data variables and does not contain any quantifiers or data terms.

Chapter 5

Instantiation of parameterised boolean equation systems

In the *mCRL2-toolset*, PBESs are used to represent a system combined with a certain property. Verification of such a property can be done by computing the solution of a PBES. At the start of the research, no PBES-solvers were present in the *mCRL2-toolset*.

A step towards solving PBES is instantiating them. This instantiation can lead to a BES, which can be solved using BES-solving techniques like gauß elimination (see [Mad97], Section 6.4). If the instantiation does not lead to a BES in a lot of cases it will lead to a PBES which predicate formulae are of reduced complexity. This can result in better performance of symbolic approximation techniques for solving PBESs.

In this chapter we discuss the research which is conducted on the instantiation of PBESs. This research has led to a tool *pbes2bes*, which implements the approaches discussed later in this chapter.

5.1 Researched approaches

Instantiating the data variables in the signature of predicate variables in a PBES can reduce those PBESs in complexity and may result in a BES. It can not be guaranteed that such an instantiation process leads to a finite (P)BES as result. For example, PBES $\mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = X(b, n+1)$, with initial state $X(\top, 0)$, would lead to an infinite BES if all data variables of predicate variable X are instantiated with every possible value those data variables can have.

We have researched two techniques to instantiate PBESs. The first technique, called the *finite approach*, eliminates all data variables in the predicate variables which are of finite type. This is done by creating a new parameterised boolean equation for each value a data variable can have. The second technique, which is called the *lazy approach*, uses techniques to create a BES, which only contains those boolean equations which are needed to solve the original PBES in the initial state.

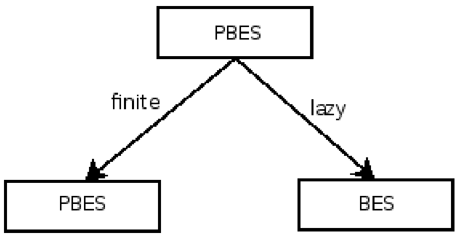

Figure 5.1: Overview of the lazy and finite approach

5.1.1 Finite approach

The finite approach is a useful technique, when a PBES has data variables which are of infinite type. These infinite data types will (possibly) lead to a computation of an infinite PBES. Take for example the following PBES: $\mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = X(b, n+1)$, with initial state $X(\top, 0)$. Instantiating all possible values for the data variables leads to an infinite PBES, because $\mathbb{N}$ is an infinite data sort. The finite approach, however, will only instantiate the data variable b, which is of data sort $\mathbb{B}$ and thus results in a PBES with the same solution, but without any finite data variables in the predicate variables of that PBES. The resulting PBES has a finite number of equations, and thus the finite approach is guaranteed to give a result in finite time.

The PBES which results from the finite approach can improve the performance of symbolic model checking techniques, because the right hand sides of the parameterised boolean equations in the PBES may be of reduced complexity. Because the finite approach does not need the initial state to compute the resulting PBES, the solution of the whole PBES can still be computed, while with the lazy approach, the focus is purely on solving the PBES in the initial state.

A disadvantage of the finite approach is that it is possible that more boolean equations are computed than needed for solving the PBES in the initial state. Also this technique will not lead to a BES in most cases.

5.1.2 Lazy approach

The lazy approach is a translation from PBESs to BESs, assuming that all quantifiers at the right hand sides of the parameterised boolean equations can be eliminated. If it is not possible to eliminate all quantifiers, the result will be a PBES, because quantifiers are not allowed in the grammar of predicate formulae of BESs.

This approach takes advantage of the fact that we want to solve the PBES in the initial state. This means we only consider those boolean equations on which the initial state depends. In many cases, this leads to a much smaller BES than the BES which could eventually be created using the finite approach.

Using this approach it is possible that a PBES with infinite data types translates to a finite BES. A typical example of such a PBES is $\mu X(n{:}\mathbb{N}) = n < 3 \wedge X(n+1)$, with initial state $X(0)$.

It is possible however, that the lazy approach leads to an infinite computation. Take the example of the finite approach: $\mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = X(b, n+1)$ with initial state $X(\top, 0)$. The initial state depends on every predicate variable instantiation where b is equal to $\top$ and n is equal to any element of $\mathbb{N}$. Because $\mathbb{N}$ is an infinite data sort, the lazy approach will not lead to a finite BES.

5.1.3 Names of new predicate variables

Both the finite and lazy approach compute new equations. Those new equations must have unique predicate variables. There are several possibilities to obtain this:

1. Add the values of the instantiated data variables to the predicate variable. The set of all possible predicate variables can be described as follows: $\{X_d | X \in \mathcal{X} \land d \in \mathit{signature}(X)\}$, where $\mathit{signature}(X)$ is the set of all possible instantiations of X. In this thesis we use this method.

2. Add a (for each predicate variable (instantiation) unique) natural number to the predicate variable (instantiation).

5.2 Finite approach: Compute an equation for each instantiation

The finite approach can reduce the complexity of a PBES by eliminating all data variables of finite data type. For each data variable of finite data type, all possible values are instantiated, resulting in as many new equations computed as there are values for a data variable. Data variables of infinite data types remain unchanged and therefore the result is a PBES.

In the finite approach, a PBES $\mathcal{E}$ is instantiated recursively over the predicate variables which occur in the left hand sides of $\mathcal{E}$. For each predicate variable X in $\mathit{lhs}(\mathcal{E})$, all equations which result from instantiating all possible values for the data variables of the finite data types in the signature of X are computed. Also, all occurrences of X in each right hand side of the equations in $\mathcal{E}$ are replaced with predicate variable instantiations, derived from X, without any finite data types. So, when instantiating a predicate variable X in an equation $\sigma Y(d{:}D, e{:}E) = \varphi$, there are two possibilities:

1. If $X = Y$, then all possible equations without any finite data types are computed and in the right hand side of each computed equation all occurrences of X are replaced with predicate variable instantiations, derived from X, without finite data types.

2. If $X \neq Y$, all occurrences of X in φ are replaced with predicate variable instantiations, derived from X, without finite data types.

Definition 5.1. Transformation of a PBES by instantiating finite data types of the signature of a predicate variable.
Before we can define a transformation function for the finite approach, we need to make a distinction between finite and infinite data types. Therefore we change the form of parameterised boolean equations to: $\sigma Y(d{:}D, e{:}E) = \varphi$, where D is a possible empty product of finite data sorts and E is a possible empty product of infinite data sorts. The definition for the transformation function becomes:

$\mathsf{ftrans}(\epsilon, X) = \epsilon$
$\mathsf{ftrans}((\sigma Y(d{:}D, e{:}E) = \varphi)\mathcal{E}, X) =$
$$\begin{cases} \{\sigma Y_{d_i}(e{:}E) = \mathsf{fsubst}(\varphi[d := d_i], X) \mid d_i \in D\}(\mathsf{ftrans}(\mathcal{E}, X)) & if\ Y = X \\ (\sigma Y(d{:}D, e{:}E) = \mathsf{fsubst}(\varphi, X))(\mathsf{ftrans}(\mathcal{E}, X)) & if\ Y \neq X \end{cases}$$

Where D is a possible empty product of finite data sorts, E is a possible empty product of infinite data sorts, $\{\sigma Y_{d_i}(e{:}E) = \mathsf{fsubst}(\varphi[d := d_i], X) | d_i \in D\}$ denotes the sequence of parameterised boolean equations with all possible instantiations for D and fsubst is defined inductively as follows:

$$\begin{aligned}
\mathsf{fsubst}(b, X) &= b \\
\mathsf{fsubst}(\varphi_1 \wedge \varphi_2, X) &= \mathsf{fsubst}(\varphi_1, X) \wedge \mathsf{fsubst}(\varphi_2, X) \\
\mathsf{fsubst}(\varphi_1 \vee \varphi_2, X) &= \mathsf{fsubst}(\varphi_2, X) \vee \mathsf{fsubst}(\varphi_2, X) \\
\mathsf{fsubst}(\forall d{:}D.\varphi, X) &= \begin{cases} \forall d{:}D.\mathsf{fsubst}(\varphi, X) & if\ D\ is\ of\ infinite\ data\ type \\ \bigwedge_{p \in D} \mathsf{fsubst}(\varphi[d := p], X) & if\ D\ is\ of\ finite\ data\ type \end{cases} \\
\mathsf{fsubst}(\exists d{:}D.\varphi, X) &= \begin{cases} \exists d{:}D.\mathsf{fsubst}(\varphi, X) & if\ D\ is\ of\ infinite\ data\ type \\ \bigvee_{p \in D} \mathsf{fsubst}(\varphi[d := p], X) & if\ D\ is\ of\ finite\ data\ type \end{cases} \\
\mathsf{fsubst}(Y(d, e), X) &= \begin{cases} \bigvee_{p \in D}(p = d \wedge X_p(e)) & if\ Y = X \\ Y(d, e) & if\ Y \neq X \end{cases}
\end{aligned}$$

Where b is a boolean value or an expression which can be rewritten to a boolean value and $\bigvee_{p \in D}(p = d \wedge X_p(e))$ represents the disjunction of all possible instantiations of the finite data sorts of the predicate variable X.

Example 5.2. Transformation function.
Take a PBES $\mathcal{E}_p =$
$\mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = (Y(b, n, n+2) \wedge b) \vee \neg b$
$\nu Y(b{:}\mathbb{B}, n, m{:}\mathbb{N}) = X(\neg b, n) \wedge Y(b, n+1, m) \wedge n \leq m$

To do a complete transformation for $\mathcal{E}_p$, the transformation function must be done for both X and Y, thus: $\mathsf{ftrans}(\ \mathsf{ftrans}(\mathcal{E}_p, X), Y\ )$. The result of this transformation does not contain any finite data sorts.

As the example shows, for a transformation of all predicate variables in the PBES, the ftrans-function has to be executed as many times as there are predicate variables in the PBES. Therefore we generalise ftrans, to do the transformation for multiple predicate variables at the same time.

Definition 5.3. Transformation for multiple variables.

$\mathsf{fgtrans}(\epsilon, \mathcal{Y}) = \epsilon$
$\mathsf{fgtrans}((\sigma Y(d{:}D, e{:}E) = \varphi)\mathcal{E}, \mathcal{Y}) =$
$$\begin{cases} \{\sigma Y_{d_i}(e{:}E) = \mathsf{fgsubst}(\varphi[d := d_i], \mathcal{Y}) \mid d_i \in D\}(\mathsf{fgtrans}(\mathcal{E}, \mathcal{Y})) & if\ Y \in \mathcal{Y} \\ (\sigma Y(d{:}D, e{:}E) = \mathsf{fgsubst}(\varphi, \mathcal{Y}))(\mathsf{fgtrans}(\mathcal{E}, \mathcal{Y})) & if\ Y \notin \mathcal{Y} \end{cases}$$

Where $\mathcal{Y}$ is the set of predicate variables for which the transformation is done, D is a possible empty product of finite data sorts, E is a possibly empty product of infinite data sorts, $\{\sigma Y_{d_i}(e{:}E) = \mathsf{fgsubst}(\varphi[d := d_i], \mathcal{Y}) | d_i \in D\}$ denotes the sequence of parameterised boolean equations with all possible instantiations for D and $\mathsf{fgsubst}$ is defined inductively as follows:

$$\begin{array}{rcl}
\mathsf{fgsubst}(b, \mathcal{Y}) & = & b \\
\mathsf{fgsubst}(\varphi_1 \wedge \varphi_2, \mathcal{Y}) & = & \mathsf{fgsubst}(\varphi_1, \mathcal{Y}) \wedge \mathsf{fgsubst}(\varphi_2, \mathcal{Y}) \\
\mathsf{fgsubst}(\varphi_1 \vee \varphi_2, \mathcal{Y}) & = & \mathsf{fgsubst}(\varphi_2, \mathcal{Y}) \vee \mathsf{fgsubst}(\varphi_2, \mathcal{Y}) \\
\mathsf{fgsubst}(\forall d{:}D.\varphi, \mathcal{Y}) & = & \begin{cases} \forall d{:}D.\mathsf{fgsubst}(\varphi, \mathcal{Y}) & \textit{if } D \textit{ is of infinite data type} \\ \bigwedge_{p \in D} \mathsf{fgsubst}(\varphi[d := p], \mathcal{Y}) & \textit{if } D \textit{ is of finite data type} \end{cases} \\
\mathsf{fgsubst}(\exists d{:}D.\varphi, \mathcal{Y}) & = & \begin{cases} \exists d{:}D.\mathsf{fgsubst}(\varphi, \mathcal{Y}) & \textit{if } D \textit{ is of infinite data type} \\ \bigvee_{p \in D} \mathsf{fgsubst}(\varphi[d := p], \mathcal{Y}) & \textit{if } D \textit{ is of finite data type} \end{cases} \\
\mathsf{fgsubst}(Y(d,e), \mathcal{Y}) & = & \begin{cases} \bigvee_{p \in D}(p = d \wedge Y_p(e)) & if\ Y \in \mathcal{Y} \\ Y(d,e) & if\ Y \notin \mathcal{Y} \end{cases}
\end{array}$$

Example 5.4. Transformation function for multiple variables at the same time.
Example 5.2 can be rewritten as far as possible using the generalized transformation function, where $\mathcal{Y}$ is the set of predicate variables X and Y:

$$\begin{array}{ll}
& \mathsf{trans}(\\
& \quad \mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = (Y(b, n, n+2) \wedge b) \vee \neg b \\
& \quad \nu Y(b{:}\mathbb{B}, n, m{:}\mathbb{N}) = X(\neg b, n) \wedge Y(b, n+1, m) \wedge n \leq m \\
& , \mathcal{Y}) \\
= & \qquad \{\text{definition } \mathsf{trans}\ \} \\
& \mu X_\perp(n{:}\mathbb{N}) = \mathsf{subst}(\ (Y(\perp, n, n+2) \wedge \perp) \vee \top, \mathcal{Y}\ ) \\
& \mu X_\top(n{:}\mathbb{N}) = \mathsf{subst}(\ (Y(\top, n, n+2) \wedge \top) \vee \perp, \mathcal{Y}\ ) \\
& \mathsf{trans}(\ \nu Y(b{:}\mathbb{B}, n, m{:}\mathbb{N}) = X(\neg b, n) \wedge Y(b, n+1, m) \wedge n \leq m,\ \mathcal{Y}\ ) \\
= & \qquad \{\text{definition subst, definition } \mathsf{trans}\ \} \\
& \mu X_\perp(n{:}\mathbb{N}) = (\ \bigvee_{p \in \mathbb{B}}(p = \perp \wedge Y_p(n, n+2)) \wedge \perp\ ) \vee \top \\
& \mu X_\top(n{:}\mathbb{N}) = (\ \bigvee_{p \in \mathbb{B}}(p = \top \wedge Y_p(n, n+2)) \wedge \top\ ) \vee \perp \\
& \nu Y_\perp(n, m{:}\mathbb{N}) = \mathsf{subst}(\ X(\top, n) \wedge Y(\perp, n+1, m) \wedge n \leq m,\ \mathcal{Y}\ ) \\
& \nu Y_\top(n, m{:}\mathbb{N}) = \mathsf{subst}(\ X(\perp, n) \wedge Y(\top, n+1, m) \wedge n \leq m,\ \mathcal{Y}\ ) \\
& \mathsf{trans}(\epsilon, \mathcal{Y}) \\
= & \qquad \{\text{definition } \bigvee, \text{ definition } \mathsf{subst}, \text{ definition } \mathsf{trans}\ \} \\
& \mu X_\perp(n{:}\mathbb{N}) = (\ (\ (\perp = \perp \wedge Y_\perp(n, n+2)) \vee (\top = \perp \wedge Y_\top(n, n+2))\ ) \wedge \perp\ ) \vee \top \\
& \mu X_\top(n{:}\mathbb{N}) = (\ (\ (\perp = \top \wedge Y_\perp(n, n+2)) \vee (\top = \top \wedge Y_\top(n, n+2))\ ) \wedge \top\ ) \vee \perp \\
& \nu Y_\perp(n, m{:}\mathbb{N}) = \bigvee_{p \in \mathbb{B}}(p = \top \wedge X_p(n)) \wedge \bigvee_{q \in \mathbb{B}}(q = \perp \wedge Y_q(n+1, m)) \wedge n \leq m \\
& \nu Y_\top(n, m{:}\mathbb{N}) = \bigvee_{p \in \mathbb{B}}(p = \perp \wedge X_p(n)) \wedge \bigvee_{q \in \mathbb{B}}(q = \top \wedge Y_q(n+1, m)) \wedge n \leq m \\
= & \qquad \{\text{definition } \bigvee\} \\
& \mu X_\perp(n{:}\mathbb{N}) = (\ (\ (\perp = \perp \wedge Y_\perp(n, n+2)) \vee (\top = \perp \wedge Y_\top(n, n+2))\ ) \wedge \perp\ ) \vee \top \\
& \mu X_\top(n{:}\mathbb{N}) = (\ (\ (\perp = \top \wedge Y_\perp(n, n+2)) \vee (\top = \top \wedge Y_\top(n, n+2))\ ) \wedge \top\ ) \vee \perp \\
& \nu Y_\perp(n, m{:}\mathbb{N}) = (\ (\perp = \top \wedge X_\perp(n)) \vee (\top = \top \wedge X_\top(n))\ ) \\
& \qquad\qquad \wedge(\ (\perp = \perp \wedge Y_\perp(n+1, m)) \vee (\top = \perp \wedge Y_\top(n+1, m))\ ) \wedge n \leq m \\
& \nu Y_\top(n, m{:}\mathbb{N}) = (\ (\perp = \perp \wedge X_\perp(n)) \vee (\top = \perp \wedge X_\top(n))\ ) \\
& \qquad\qquad \wedge(\ (\perp = \top \wedge Y_\perp(n+1, m)) \vee (\top = \top \wedge Y_\top(n+1, m))\ ) \wedge n \leq m
\end{array}$$

Using techniques from [GrW05] and algebraic and logical reasoning, this can be simplified to:
$\mu X_\perp(n{:}\mathbb{N}) = \top$
$\mu X_\top(n{:}\mathbb{N}) = Y_\top(n, n+2)$
$\nu Y_\perp(n, m{:}\mathbb{N}) = X_\top(n) \wedge Y_\perp(n+1, m) \wedge n \leq m$
$\nu Y_\top(n, m{:}\mathbb{N}) = X_\perp(n) \wedge Y_\top(n+1, m) \wedge n \leq m$

The instantiation using the finite approach preserves the solution of the PBES, which is proven in [DPW07]. We will look again at example 5.4. The original PBES $\mathcal{E}$ was:
$\mu X(b{:}\mathbb{B}, n{:}\mathbb{N}) = (Y(b, n, n+2) \wedge b) \vee \neg b$
$\nu Y(b{:}\mathbb{B}, n, m{:}\mathbb{N}) = X(\neg b, n) \wedge Y(b, n+1, m) \wedge n \leq m$
The resulting PBES $\mathcal{E}_f$ became:
$\mu X_{\perp}(n{:}\mathbb{N}) = \top$
$\mu X_{\top}(n{:}\mathbb{N}) = Y_{\top}(n, n+2)$
$\nu Y_{\perp}(n, m{:}\mathbb{N}) = X_{\top}(n) \wedge Y_{\perp}(n+1, m) \wedge n \leq m$
$\nu Y_{\top}(n, m{:}\mathbb{N}) = X_{\perp}(n) \wedge Y_{\top}(n+1, m) \wedge n \leq m$
As we can see, the equation with predicate variable $X(b{:}\mathbb{B}, n{:}\mathbb{N})$ in $\mathcal{E}$ resulted in $X_{\perp}(n{:}\mathbb{N})$ and $X_{\top}(n{:}\mathbb{N})$ in $\mathcal{E}_f$. Thus, for an equation in $\mathcal{E}_f$ it can easily be found from which predicate variable in $\mathcal{E}$ it is derived.

The number of equations in the resulting PBES with respect to the original PBES, depends on both the number of finite data types and the number of data elements in those finite data types of the predicate variables. For example, if a PBES has one equation with a predicate variable $X(b, c{:}\mathbb{B})$, the resulting PBES will have four equations, because both data variables in X are of data type $\mathbb{B}$, which has two data elements.

5.3 Lazy approach: Only compute needed equations

As the finite approach does not lead to BESs if the original PBES contains infinite data types in its predicate variables, there is a need for an approach which can result in BESs in cases with data variables of infinite data type. Therefore we researched the lazy approach, which assumes we want to solve a PBES in its initial state. In this approach, we only consider those equations the initial state depends on.

To compute a BES out of a PBES $\mathcal{E}$ using the lazy approach, we start with the initial state $X(d_{init})$ of $\mathcal{E}$. For $X(d_{init})$ we compute the equation $\sigma X_{d_{init}} = \varphi[d := d_{init}]$. From this equation we determine all predicate variable instantiations that occur in $\varphi[d := d_{init}]$; we say $X_{d_{init}}$ depends on those predicate variable instantiations. For each predicate variable instantiation the initial state depends on, an equation is computed and all predicate variable instantiations which the equation depends on are computed recursively. When all equations are computed, the resulting equation system has to be constructed in such a way that the order of the equations in $\mathcal{E}$ is respected. Formalising, this leads to the following transformation function:

Definition 5.5. Lazy transformation function
To compute a BES out of a PBES $\mathcal{E}$, using the lazy approach, we define:
$\mathsf{ltrans}(X(d_{init}), \mathcal{E}) = \mathsf{sort}(\mathsf{ltrans}'(\{X(d_{init})\}, \emptyset, \mathcal{E}), \mathcal{E})$
Where $X(d_{init})$ is the initial state of the PBES and ltrans' needs parameters which contains the set of predicate variable instantiations which have to be done, the set of predicate variable instantiations which are done, and the original PBES. It is defined inductively as:

$\mathsf{ltrans}'(\emptyset, done, \mathcal{E}) = \epsilon$
$\mathsf{ltrans}'(\{X(v)\} \cup todo, done, \mathcal{E}) =$
$\qquad \mathsf{pbecreate}(X(v), \mathcal{E})\ \mathsf{ltrans}'(\ (todo \cup \mathsf{dep}(X(v), \mathcal{E})) \backslash (done \cup \{X(v)\}), done \cup \{X(v)\}, \mathcal{E}\ )$

Where $X(v)$ is an arbitrary predicate variable instantiation.

The function sort sorts all the boolean equations with respect to the order of the parameterised boolean equations in the original PBES and is defined inductively as follows:

$\mathsf{sort}(\mathcal{E}, \epsilon) = \epsilon$
$\mathsf{sort}(\mathcal{E}, (\sigma X(d : D) = \varphi)\mathcal{E}_1) =$
$\qquad (\{(\sigma X_n = \varphi) \mid (\sigma X_n = \varphi) \in \mathcal{E} \wedge X_n \in \{X_d \mid X \in \mathcal{X} \wedge d \in signature(X)\}\ \})\ \mathsf{sort}(\mathcal{E}, \mathcal{E}_1)$

The function dep determines the set of predicate variable instantiations a given predicate variable instantiation depends on and is defined inductively as:

$$\begin{array}{rcl}
\mathsf{dep}(X(d),\epsilon) & = & \emptyset \\
\mathsf{dep}(X(d),(\sigma Y(e{:}E)=\varphi)\mathcal{E}') & = & \begin{cases}\mathsf{instset}(\varphi[e:=d]) & \mathit{if}\ X=Y\\ \mathsf{dep}(X(d),\mathcal{E}') & \mathit{if}\ X\neq Y\end{cases}
\end{array}$$

Where instset is defined inductively as follows:

$$\begin{array}{rcl}
\mathsf{instset}(b) & = & \emptyset \\
\mathsf{instset}(\varphi_1\wedge\varphi_2) & = & \mathsf{instset}(\varphi_1)\cup\mathsf{instset}(\varphi_2) \\
\mathsf{instset}(\varphi_1\vee\varphi_2) & = & \mathsf{instset}(\varphi_1)\cup\mathsf{instset}(\varphi_2) \\
\mathsf{instset}(\forall d{:}D.\varphi) & = & \bigcup\limits_{v\in D}\mathsf{instset}(\varphi[d:=v]) \\
\mathsf{instset}(\exists d{:}D.\varphi) & = & \bigcup\limits_{v\in D}\mathsf{instset}(\varphi[d:=v]) \\
\mathsf{instset}(X(d)) & = & \{X(d)\}
\end{array}$$

The function pbecreate computes a boolean equation from an $\mathcal{E}$ and a predicate variable instantiation $X(v)\in\mathcal{E}$ and is defined inductively as:

$$\begin{array}{rcl}
\mathsf{pbecreate}(X(v),\epsilon) & = & \epsilon \\
\mathsf{pbecreate}(X(v),(\sigma Y(d{:}D)=\varphi)\mathcal{E}') & = & \begin{cases}\sigma X_v=\mathsf{lsubst}(\varphi[d:=v],\mathcal{X}) & \mathit{if}\ X=Y\\ \mathsf{pbecreate}(X(v),\mathcal{E}') & \mathit{if}\ X\neq Y\end{cases}
\end{array}$$

Where lsubst is defined inductively as:

$$\begin{array}{rcl}
\mathsf{lsubst}(b,\mathcal{X}) & = & b \\
\mathsf{lsubst}(\varphi_1\wedge\varphi_2,\mathcal{X}) & = & \mathsf{lsubst}(\varphi_1,\mathcal{X})\wedge\mathsf{lsubst}(\varphi_2,\mathcal{X}) \\
\mathsf{lsubst}(\varphi_1\vee\varphi_2,\mathcal{X}) & = & \mathsf{lsubst}(\varphi_2,\mathcal{X})\vee\mathsf{lsubst}(\varphi_2,\mathcal{X}) \\
\mathsf{lsubst}(\forall d{:}D.\varphi,\mathcal{X}) & = & \forall d{:}D.\mathsf{lsubst}(\varphi,\mathcal{X}) \\
\mathsf{lsubst}(\exists d{:}D.\varphi,\mathcal{X}) & = & \exists d{:}D.\mathsf{lsubst}(\varphi,\mathcal{X}) \\
\mathsf{lsubst}(Y(d,e),\mathcal{X}) & = & \bigvee\limits_{p\in D}(p=d\wedge Y_p)
\end{array}$$

Where $\mathcal{X}$ is the set of all predicate variables.

Example 5.6. To show how the lazy approach instantiates a PBES we consider the following PBES:
$\mu X(b{:}\mathbb{B}) = Y(\bot) \wedge X(b)$
$\nu Y(b{:}\mathbb{B}) = X(b)$
with initial state $X(\top)$. We will denote $\mu X(b{:}\mathbb{B}) = Y(\bot) \wedge X(b)$ as σ_1 and $\nu Y(b{:}\mathbb{B}) = X(b)$ as σ_2, and the complete system as $\sigma_1\sigma_2$.

Using the lazy approach we can rewrite this to a BES. Some steps in the example are combined, and the use of the inductive definitions is assumed to be clear. In appendix C the example is shown step by step.

$$\begin{array}{ll}
 & \mathsf{ltrans}(\ X(\top), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{ltrans}\ \} \\
 & \mathsf{sort}(\ \mathsf{ltrans}'(\ \{X(\top)\}, \emptyset, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{ltrans}'\} \\
 & \mathsf{sort}(\ (\mathsf{pbecreate}(X(\top), \sigma_1\sigma_2))\ (\mathsf{ltrans}'(\ (\emptyset \cup \mathsf{dep}(X(\top), \sigma_1\sigma_2)) \backslash (\emptyset \cup \{X(\top)\}), \emptyset \cup \{X(\top)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{pbecreate}\text{, definition } \mathsf{dep}\text{, algebra}\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\mathsf{ltrans}'(\ \{Y(\bot)\}, \{X(\top)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{ltrans}'\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\mathsf{pbecreate}(Y(\bot), \sigma_1\sigma_2) \\
 & \qquad (\mathsf{ltrans}'(\ (\emptyset \cup \mathsf{dep}(Y(\bot), \sigma_1\sigma_2)) \backslash (\{X(\top)\} \cup \{Y(\bot)\}), \{X(\top)\} \cup \{Y(\bot)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{pbecreate}\text{, definition } \mathsf{dep}\text{, algebra}\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mathsf{ltrans}'(\ \{X(\bot)\}, \{X(\top), Y(\bot)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{ltrans}'\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mathsf{pbecreate}(X(\bot), \sigma_1\sigma_2)) \\
 & \qquad (\mathsf{ltrans}'(\ (\emptyset \cup \mathsf{dep}(X(\bot), \sigma_1\sigma_2)) \backslash (\{X(\top), Y(\bot)\} \cup \{X(\bot)\}), \{X(\top), Y(\bot)\} \cup \{X(\bot)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{pbecreate}\text{, definition } \mathsf{dep}\text{, algebra}\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mu X_\bot = Y_\bot \wedge X_\bot)\ (\mathsf{ltrans}'(\ (\emptyset, \{X(\top), Y(\bot), X(\bot)\}, \sigma_1\sigma_2\ ), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{ltrans}'\} \\
 & \mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mu X_\bot = Y_\bot \wedge X_\bot), \sigma_1\sigma_2\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{sort}\ \} \\
 & (\mu X_\top = Y_\bot \wedge X_\top)\ (\mu X_\bot = Y_\bot \wedge X_\bot) \\
 & \qquad (\mathsf{sort}(\ (\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mu X_\bot = Y_\bot \wedge X_\bot), \sigma_2)\ ) \\
= & \qquad\qquad \{\text{Definition } \mathsf{sort}\ \} \\
 & (\mu X_\top = Y_\bot \wedge X_\top)\ (\mu X_\bot = Y_\bot \wedge X_\bot)\ (\nu Y_\bot = X_\bot) \\
 & \qquad (\mathsf{sort}((\mu X_\top = Y_\bot \wedge X_\top)\ (\nu Y_\bot = X_\bot)\ (\mu X_\bot = Y_\bot \wedge X_\bot), \epsilon)) \\
= & \qquad\qquad \{\text{Definition } \mathsf{sort}\ \} \\
 & (\mu X_\top = Y_\bot \wedge X_\top)\ (\mu X_\bot = Y_\bot \wedge X_\bot)\ (\nu Y_\bot = X_\bot)
\end{array}$$

The lazy approach preserves the solution of the PBES *in the initial state.*
The number of equations in the resulting PBES depends on the number of equations the initial state of the original PBES depends on. For example, the PBES in example 5.6 leads to a BES with three equations.

5.4 Implementation

The finite and lazy approach are implemented in the tool *pbes2bes*. The transformation and substitution functions are implemented straightforward using lists of equations and the *PBES-library*. This library contains functions to read, save and manipulate parts of a PBES in terms of the internal format (see appendix A for an overview of the internal format). In the latter of this section, we discuss some of the difficulties experienced during the implementation.

Free variables In a PBES, free variables may be present. Free variables are data variables in a parameterised boolean equation which do not occur on the left hand side of that parameterised boolean equation and are not bound by a quantifier. Such a free variable may be instantiated by the data environment, but because the variable is free, the value which is instantiated for the free variable is not of interest. Therefore we instantiate it with a random value.
A function *free_variables* has been added to the library, which returns a *data_variable_list*, containing all free variables in the PBES. All free data variables are replaced with a random instantiation of the data sort of the variable. For this replacement a slightly changed variant of an existing function for instantiating a variable by a random instantiation is used. This function (called Find_Dummy) could be rewritten to be part of the *PBES-library* so other tools can use this functionality too.
To give the user control on which values has to be instantiated, it is possible to implement a function which asks the user to choose the instantiation he wants to use for the free variable.

Renaming predicate variable (instantiations) All newly computed predicate variables and predicate variable instantiations must be unique for the PBES. This is achieved by adding all pretty printed values of the data elements in the predicate variable (instantiation) to a unique predicate variable (instantiation), each value separated by the @-symbol. So if a predicate variable instantiation $X(true, 7)$ occurs, it will be rewritten to $X@true@7$. Data variables which are of infinite data type are in the finite approach not added to the predicate variable, but remain in the signature of the predicate variable (instantiation).

Rewriting of predicate formulae and data terms In the libraries of mCRL2 a data rewriter is already available. However this data rewriter is not capable of dealing with predicate formulae. Therefore a *PBES-rewriter* is written, which rewrites a predicate formula φ to a (possibly) smaller form ψ which is equivalent to φ. Recall the grammar of φ:

$$\varphi ::= \ b \mid \top \mid \ \bot \ \mid X(e) \mid \varphi \ \wedge \ \psi \mid \varphi \ \vee \ \psi \mid \forall \ d{:}D.\varphi \mid \exists \ d{:}D.\varphi$$

Where b is a data term of data sort $\mathbb{B}$, $\top$ and $\bot$ are elements of data sort $\mathbb{B}$, X is a predicate variable, e consists of zero or more data sorts and d is a data variable of sort D.

We will define the *PBES-rewriter* inductively using a function pbesr denoting the *PBES-rewriter* and datar denoting the data rewriter.

$$
\begin{array}{rcl}
\mathsf{pbesr}(b) & = & \mathsf{datar}(b) \\
\mathsf{pbesr}(\top) & = & \top \\
\mathsf{pbesr}(\bot) & = & \bot \\
\mathsf{pbesr}(\varphi \wedge \psi) & = & \begin{cases} \bot & \textit{if } \mathsf{pbesr}(\varphi) = \bot \vee \mathsf{pbesr}(\psi) = \bot \\ \mathsf{pbesr}(\psi) & \textit{if } \mathsf{pbesr}(\varphi) = \top \\ \mathsf{pbesr}(\varphi) & \textit{if } \mathsf{pbesr}(\psi) = \top \\ \mathsf{pbesr}(\varphi) & \textit{if } \mathsf{pbesr}(\varphi) = \mathsf{pbesr}(\psi) \\ \mathsf{pbesr}(\varphi) \wedge \mathsf{pbesr}(\psi) & \textit{otherwise} \end{cases} \\
\mathsf{pbesr}(\varphi \vee \psi) & = & \begin{cases} \top & \textit{if } \mathsf{pbesr}(\varphi) = \top \vee \mathsf{pbesr}(\psi) = \top \\ \mathsf{pbesr}(\psi) & \textit{if } \mathsf{pbesr}(\varphi) = \bot \\ \mathsf{pbesr}(\varphi) & \textit{if } \mathsf{pbesr}(\psi) = \bot \\ \mathsf{pbesr}(\varphi) & \textit{if } \mathsf{pbesr}(\varphi) = \mathsf{pbesr}(\psi) \\ \mathsf{pbesr}(\varphi) \vee \mathsf{pbesr}(\psi) & \textit{otherwise} \end{cases} \\
\mathsf{pbesr}(\forall\ d{:}D.\varphi) & = & \begin{cases} \mathsf{pbesr}(\varphi) & \textit{if } d \textit{ does not occur in } \varphi \\ \mathsf{pbesr}(\bigwedge_{e:D} \varphi[d := e]) & \textit{if } D \textit{ is of finite data type} \\ \forall\ d{:}D.\mathsf{pbesr}(\varphi) & \textit{otherwise} \end{cases} \\
\mathsf{pbesr}(\exists\ d{:}D.\varphi) & = & \begin{cases} \mathsf{pbesr}(\varphi) & \textit{if } d \textit{ does not occur in } \varphi \\ \mathsf{pbesr}(\bigvee_{e:D} \varphi[d := e]) & \textit{if } D \textit{ is of finite data type} \\ \exists\ d{:}D.\mathsf{pbesr}(\varphi) & \textit{otherwise} \end{cases} \\
\mathsf{pbesr}(X(e)) & = & X(\mathsf{datar}(e))
\end{array}
$$

This *PBES-rewriter* has been implemented straightforward from the definition, using the *PBES-library.*

Saving in other formats There are several tools which can solve BESs, like the BES-solver in the *CADP-toolset.* If the result of the lazy approach is a BES, it is possible to store the BES in the so called CWI-format. This format is used at the *Centrum voor Wiskunde en Informatica* in the BES-solvers they are developing. The CWI format is implemented by first creating a hash-table which contains each predicate variable indexed with a natural number. Each equation is rewritten to the CWI format where each predicate variable is replaced with the natural number in the hash-table. This transformation to the CWI-format, after the computation of the BES has finished.
Another format which is used, is the format used in the CADP toolset. This format is currently not implemented. Both formats are described in appendix A.

Finite approach: Enumerate finite sorts Enumeration of finite sorts is used in the finite approach, to be able to create all possible instantiations for data sorts. It is used in the ftrans and fgtrans functions. For each constructor of a given sort, a data_expression_list containing all possible instantiations is created by enumerating over all possible instantiations of the constructor.

5.5 Optimisations

The finite and lazy approach can be optimised to achieve a better performance or being able to create a BES for a larger subset of PBESs. In this section we will discuss a number of optimisations which we observed. The first two optimisations are implemented, the other optimisations are not yet implemented. All optimisations which increase the performance of the tool preserve the behavior of the definitions.

fgtrans and fgsubst functions When choosing for $\mathcal{Y}$ exactly the set of predicate variables present in a PBES $\mathcal{E}$, both in trans and subst the cases where $Y \notin \mathcal{Y}$ are never executed. Therefore these cases are not implemented in the finite approach. When there is a need to implement the case where $Y \notin \mathcal{Y}$, a set with the predicate variables for which the transformation is done must be present. The cases can be implemented straightforward.

fgsubst and lsubst functions: $\bigvee p \in D\ (p = d \wedge Y_p)$ Both in the finite and the lazy approach in the subst function predicate variable instantiations are rewritten to the form $\bigvee p \in D\ (p = d \wedge Y_p)$. The conjunction will be $\perp$ in all cases that $p \neq d$. Therefore the only case of interest is that where $p = d$. In the implementation only the predicate variable Y_p is created for the case that $p = d$, using the predicate variable renaming function.

Combining rewriting and substitution Rewriting of and substitution on predicate formulae can take a lot of time. Currently, if a data variable has to be assigned a certain value, the substitution is done in the whole predicate formula at once, before the rewriting is done. In the case of conjunctions, disjunctions and quantifier-formulae which can be rewritten to finite conjunctions or finite disjunctions, it is possible that the substitution is done on parts of a predicate formula which do not have any influence on the solution of that predicate formula.
An example of such a predicate formula is $n > 3 \wedge \varphi$, where φ is a complex predicate formula with a number of occurrences of n. If n must be substituted by 2, in the current implementation the substitution and rewrite process is done as follows:

$$\begin{array}{ll} & n > 3 \wedge \varphi \\ = & \qquad \{\text{Substitution of } n \text{ by } 2\} \\ & 2 > 3 \wedge \varphi[n := 2] \\ = & \qquad \{\text{Rewrite left hand side of and-connective}\} \\ & \perp \wedge \varphi[n := 2] \\ = & \qquad \{\text{Rewrite and-connective}\} \\ & \perp \end{array}$$

By combining substitution and rewriting, substitution on a part of a predicate formula is only done if the part is needed for the solution of that predicate formula. For the example, the process is done as follows:

$$\begin{array}{ll} & n > 3 \wedge \varphi \\ = & \qquad \{\text{Substitution of } n \text{ by } 2 \text{ in } n > 3\} \\ & 2 > 3 \wedge \varphi \\ = & \qquad \{\text{Rewrite left hand side of and-connective}\} \\ & \perp \wedge \varphi \\ = & \qquad \{\text{Rewrite and-connective}\} \\ & \perp \end{array}$$

This approach can result in a big gain in performance, if the predicate formulae in the PBES are complex and parts of it are not needed for the solution of the predicate formulae.

Removing quantifiers in pbesr Currently removing quantifiers is only possible if the data variables of the quantifiers are not used in the quantifier expression, or if all data variables are of finite data sort. There are, however, more possibilities to remove a quantifier, which will be discussed here.

If a universal quantification is of the form $\forall d{:}D.condition \Rightarrow \varphi$, the result of an instantiation of d is always $\top$ if the *condition* is $\bot$. In the other cases the result of the expression depends on φ. This results in a conjunction of all cases where the *condition* is $\bot$, which can be a finite conjunction.

Dually the same is the case if an existential quantification is of the form $\exists d{:}D.condition \wedge \varphi$, where the result of the quantification is the disjunction of all cases where *condition* is $\top$, which can be a finite disjunction.

A third way to remove more quantifications, is to use theorem 6.2 of [GrW05a]. This theorem states that if the expressions of quantifiers are of the form ITE(d=e, ψ, χ) or nested ITE's on the last argument of the ITE (where ITE must be read as an $if-then-else$ construct) the quantifiers can be replaced by a conjunction in case of a universal quantifier and by a disjunction in case of an existential quantifier.

Ordering of equations in the lazy approach The sort function is implemented as follows: For each predicate variable X in the original PBES $\mathcal{E}$, the result is checked equation by equation and it is checked if the predicate variable instantiations is derived from X, by checking if the predicate variable until the first occurrence of the @-symbol is equal to X. If this is the case, the equation is added to the final result, otherwise it is added to the set of equations which have to be sorted.

As all predicate variable names are checked if they are derived from X, an optimisation is possible: create (for each equation in $\mathcal{E}$) a tuple $(X, \{\sigma Y_n = \varphi \mid Y(n) \wedge n \in signature(Y) \wedge Y = X\})$, where $X \in lhs(\mathcal{E})$. Each equation which is computed in the lazy approach is added to the tuple containing the predicate variable the equation is derived from. Sorting of the resulting system can be defined as follows:

$$\begin{array}{rcl} \mathsf{sort}(todo, \epsilon) & = & \epsilon \\ \mathsf{sort}(todo, (\sigma X(d:D) = \varphi)\mathcal{E}') & = & (\ \mathsf{sort}'(todo, X)\ )\ (\ \mathsf{sort}(todo, \mathcal{E}')\ ) \end{array}$$

Where sort' is defined as:

$$\begin{array}{rcl} \mathsf{sort}'(\emptyset, X) & = & \epsilon \\ \mathsf{sort}'(\{(Y, \mathcal{E}_p)\} \cup todo, X) & = & \begin{cases} \mathcal{E}_p & if\ X = Y \\ \mathsf{sort}'(todo, X) & if\ X \neq Y \end{cases} \end{array}$$

Chapter 6

Case studies

To show the main differences between the finite and lazy approach, we will describe case studies on the alternating bit protocol and Lamport's bakery protocol. The alternating bit protocol is a protocol with a finite state space, where we can show that the lazy algorithm can compute smaller BESs using the lazy approach, then when we would use the finite approach. Lamport's bakery protocol has an infinite state space. A lot of properties on this system will lead to an infinite BES using the lazy approach. There are, however, properties which will lead to a finite BES.

6.1 Alternating bit protocol

The alternating bit protocol [BaW90] ensures the successful transmission of data through an unreliable channel, under the assumption that data can be resent an unlimited number of times. The protocol is shown in figure 6.1.

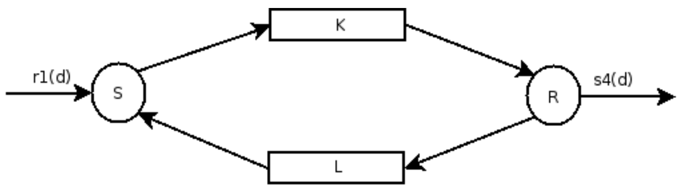

Figure 6.1: Alternating bit protocol

First, we will explain the alternating bit protocol informally. The sender (S) reads a message from r1, adds a bit to it, and sends it repeatedly to the receiver (R) through the unreliable channel K, until it receives an acknowledgement with the bit value added by the sender. the receiver reads messages from channel K. If a message is read by the receiver, it checks if the bit has the correct value. If that is the case, it sends the message (excluding the added bit) to s4, and sends the bit to the unreliable channel L. The receiver keeps sending this bit until it receives a message with the bit-value alternated to the one it is sending. When a message is sent to s4 and the sender has received an acknowledgement, the sender reads the next value from r1 and inverts the bit value and send this to the receiver.

The mCRL2-specification we use is the one which can be found in the mCRL2 toolset and is as follows:

```
sort
  D     = struct d1 | d2;
  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)
    )
  );
```

Where `c2`, `c3`, `c5` and `c6` are communicating actions which represent the process of sending and receiving between the four components of the alternating bit protocol. The action `i` is used for a non-deterministic choice between sending a message, or an error (because K and L are unreliable channels).

The properties we will look into for the alternating bit protocol are:

ABP1	No deadlock can occur
ABP2	If a message is sent, it is possible that it is read
ABP3	If a message is sent, it will be read eventually, if it isn't lost permanently
ABP4	No miracles: Messages are not created by the protocol

For each property a PBES is created, which we will identify by the number of the property. The finite and lazy approach will be used for every PBES.

To illustrate the size of the resulting equation systems, in each PBES we also use the tool *mcrl22lps* with the option -w, which replaces infinite data sorts by enumerated sorts where possible. These enumerated sorts are of finite type. The number of elements in that sort can be derived from the name of the sort. The sort has the name `Enum`n, where n is the number of elements in the sort.

For each property we will show the μ-calculus-formula in the syntax that is used by the tool *lps2pbes*, the signature of the predicate variables of the resulting PBES (obtained using the tool *pbesinfo*, see chapter 4), the number of equations using the different approaches (which is computed from the signature for the PBES in the finite approach and obtained using the tool *pbesinfo* for the lazy approach), and the time in seconds it took to compute the results for the different approaches[1]. Using a BES-solver, we solve the resulting BESs to check if the property holds on the alternating bit protocol.

ABP1 The μ-calculus formula, which describes that no deadlock can occur, is the following:

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

The PBES for ABP1 can be found in appendix D.1 and consist of one parameterised boolean equation with predicate variable X. The signature of X is the following:

```
X :: Pos x D x Bool x Pos x D x Bool x Pos x
     Bool x Pos x D x Bool -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum3 x D x Bool x Enum4 x D x Bool x Enum4 x
     Bool x Enum4 x D x Bool -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	74	74	128	24576
Time	0.20	0.15	0.57	43.53

Table 6.1: Results for ABP1

The lazy approach (with or without using enumerations) and the finite approach using enumerations result in a BES, so each predicate variable is of type `Bool`. In the finite approach using data variables of type `Pos`, the result is a PBES and all predicate variables are derived from X and have the following signature:

```
X :: Pos x Pos x Pos x Pos -> Bool
```

The property holds for the alternating bit protocol.

[1]For the computations a dual core Pentium D 3 GHz with 1 GB of memory was used.

ABP2 The property looks like this:

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

The PBES for ABP2 can be found in appendix D.2 and consist of two parameterised boolean equations with predicate variables X and Y. The signature of X and Y is the following:

```
X :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool -> Bool
Y :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool x D -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool -> Bool
Y :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool x D -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	110	110	384	73728
Time	0.25	0.18	1.44	113.69

Table 6.2: Results for ABP2

The lazy approach (with or without using enumerations) and the finite approach using enumerations result in a BES, so each predicate variable is of type `Bool`. In the finite approach using data variables of type `Pos`, the result is a PBES and all predicate variables which are derived from X and Y have the following signature:

```
X :: Pos x Pos x Pos x Pos -> Bool
Y :: Pos x Pos x Pos x Pos -> Bool
```

The property holds for the alternating bit protocol.

ABP3 The property is like ABP2, but with a fairness constraint and looks like this:

```
nu X. (
        [true]X
    &&
        forall dd:D. ([r1(dd)]( nu Y. mu Z. ([(!s4(dd)) && (!i)]Z && [i]Y) ) )
      )
```

The PBES for ABP3 can be found in appendix D.3 and consist of three parameterised boolean equations with predicate variables X, Y and Z. The signature of X, Y and Z is the following:

```
X :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool -> Bool
Y :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool x D -> Bool
Z :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool x D -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool -> Bool
Y :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool x D -> Bool
Z :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool x D -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	130	130	640	122880
Time	0.25	0.19	1.46	131.26

Table 6.3: Results for ABP4

The lazy approach (with or without using enumerations) and the finite approach using enumerations result in a BES, so each predicate variable is of type `Bool`. In the finite approach using data variables of type `Pos`, the result is a PBES and all predicate variables which are derived from X and Y have the following signature:

```
X :: Pos x Pos x Pos x Pos -> Bool
Y :: Pos x Pos x Pos x Pos -> Bool
Y :: Pos x Pos x Pos x Pos -> Bool
```

The property holds for the alternating bit protocol.

ABP4 The μ-calculus formula which describes that no messages are created in the protocol, is the following:

```
nu X. ( forall dd:D. ([!r1(dd)]X && [s4(dd)]false ) )
```

The PBES for ABP4 can be found in appendix D.4 and consist of one parameterised boolean equation with predicate variable X. The signature of X is the following:

```
X :: Pos x D x Bool x Pos x D x Bool x
     Pos x Bool x Pos x D x Bool -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum3 x D x Bool x Enum4 x D x Bool x
     Enum4 x Bool x Enum4 x D x Bool -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	74	74	128	24576
Time	0.16	0.13	0.70	37.12

Table 6.4: Results for ABP4

The lazy approach (with or without using enumerations) and the finite approach using enumerations result in a BES, so each predicate variable is of type `Bool`. In the finite approach using data variables of type `Pos`, the result is a PBES and all predicate variables are derived from X and have the following signature:
`X :: Pos x Pos x Pos x Pos -> Bool`

The property holds for the alternating bit protocol.

Rewriting of quantifiers Using PBES ABP1, we will show some of the techniques for removing quantifiers.

In ABP1 the following quantifier-expression is present: $\exists\ d0\_00{:}D.\ (s30\ ==\ 1)$. From this expression the quantifier can be removed, because the variable $e0\_00$ is not used in the expression.

Another expression which is present is:
$\exists\ e5\_00{:}\mathbb{B}.\ (\ s31 == 3 \wedge s33 == 1 \wedge \mathit{if}(e5\_00, b2, !b2) == b4\ )$.
Because $\mathbb{B}$ is a finite sort, the quantifier can be removed:
$(\ s31 == 3 \wedge s33 == 1 \wedge \mathit{if}(\top, b2, !b2) == b4\ )\ \vee\ (\ s31 == 3 \wedge s33 == 1 \wedge \mathit{if}(\bot, b2, !b2) == b4\ )$
which can be simplified to
$(\ s31 == 3 \wedge s33 == 1 \wedge b2 == b4)\ )\ \vee\ (\ s31 == 3 \wedge s33 == 1 \wedge !b2 == b4\ )$.

Conclusion The alternating bit protocol is a good example of a specification for which a BES can be computed using the lazy approach. The finite approach can lead to BESs also, but these are way bigger, in the case of the alternating bit protocol up to 1000 times.

6.2 Lamport's bakery protocol

Lamport's Bakery Protocol [Ray86] is described as follows. A process, which is waiting to enter it's critical section, chooses a number, larger than any number already chosen. Processes with a lower number are allowed to enter the critical before processes with a higher number. Because the number which is chosen can grow indefinitely, the state space of this protocol is infinite.

The mCRL2-specification of Lamport's Bakery Protocol can be found in the mCRL2 toolset and is as follows:

```
act
  send,get,c:          Bool # Nat;
  request,enter,leave: Bool;

proc P(b:Bool)        = request(b).P0(b,0) + send(b,0).P(b);

     P0(b:Bool,n:Nat) = (sum m:Nat. get(!b,m).P1(b,m + 1)) + send(b,n).P0(b,n);

     P1(b:Bool,n:Nat) =
        (sum m:Nat. get(!(b),m).
          ((n < m || m == 0) -> C1(b,n) +
                (m <= n && m != 0) -> P1(b,n))) + send(b,n).P1(b,n);

     C1(b:Bool,n:Nat) = enter(b).C2(b,n) + send(b,n).C1(b,n);

     C2(b:Bool,n:Nat) = leave(b).P(b) + send(b,n).C2(b,n);

init
  allow({request,enter,leave,c},
    comm({get|send->c}, P(true) || P(false))
  );
```

The properties we will look into for Lamport's bakery protocol are:

BAK1	No deadlock can occur
BAK2	All processes requesting a number can eventually enter the critical section
BAK3	All processes requesting a number inevitably enter the critical section
BAK4	A process inevitably enters a critical section

For each property a PBES is created, which we will identify by the number of the property. The finite and lazy approach will be used on every PBES.

To illustrate the size of the resulting equation systems, in each PBES we also use the tool *mcrl22lps* with the option -w, which replaces infinite data sorts by enumerated sorts where possible. These enumerated sorts are of finite type. The number of elements in that sort can be derived from the name of the sort. The sort has the name Enum$n$, where n is the number of elements in the sort.

For each property we will show the μ-calculus-formula, the signature of the predicate variables of the resulting PBES (obtained using the tool *pbesinfo*, the number of equations using the different approaches (which is computed from the signature for the PBES in the finite approach and obtained using the tool *pbesinfo* for the lazy approach), and the time in seconds it took to compute the results for the different approaches[2].

BAK1 The μ-calculus formula which describes that no deadlock can occur, is the following:

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

The PBES for BAK1 can be found in appendix D.5 and consist of one parameterised boolean equation with predicate variable X. The signature of X is the following:

```
X :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	-	-	4	144
Time	-	-	0.13	0.39

Table 6.5: Results for BAK1

The lazy approach leads to an infinite computation because the data variables of type `Nat` can grow indefinitely. The finite approach also can not instantiate the data variables of type `Nat` and therefore a PBES is returned. The signature for each data variable when using the finite approach without enumerations is:

```
X :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
```

When using enumerations, the signature becomes:

```
X :: Nat x Nat x Nat x Nat -> Bool
```

The property can not be checked using BES-solvers, because the result is a PBES and not a BES.

BAK2 The μ-calculus formula is the following:

```
nu X. (
        [true]X
    &&
        forall b:B. ([request(b)]( mu Y. (<true>Y || <enter(b)>true) ) )
      )
```

The PBES for BAK2 can be found in appendix D.6 and consist of two parameterised boolean equations with predicate variables X and Y. The signature of X and Y are the following:

```
X :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat -> Bool
Y :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat x Bool -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat -> Bool
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat x Bool-> Bool
```

[2]For the computations a dual core Pentium D 3 GHz with 1 GB of memory was used.

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	lazy	lazy enum	finite	finite enum
# equations	-	-	12	432
Time	-	-	0.22	0.91

Table 6.6: Results for BAK2

The lazy approach leads to an infinite computation because the data variables of type `Nat` can grow indefinitely. The finite approach also can not instantiate the data variables of type `Nat` and therefore a PBES is returned. The signature for each data variable when using the finite approach without enumerations is:

```
X :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
Y :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
```

When using enumerations, the signature becomes:

```
X :: Nat x Nat x Nat x Nat -> Bool
Y :: Nat x Nat x Nat x Nat -> Bool
```

The property can not be checked using BES-solvers, because the result is a PBES and not a BES.

BAK3 The μ-calculus formula is the following:

```
nu X. (
        [true]X
    &&
        forall b:B. ([request(b)]
                       ( mu Y. ( ([true]Y && <true>true) || <enter(b)>true) ) )
      )
```

The PBES for BAK3 can be found in appendix D.7 and consist of two parameterised boolean equations with predicate variables X and Y. The signature of X and Y are the following:

```
X :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat -> Bool
Y :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat x Bool -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat -> Bool
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat x Bool-> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	lazy	lazy enum	finite	finite enum
# equations	-	-	12	432
Time	-	-	0.29	1.15

Table 6.7: Results for BAK3

The lazy approach leads to an infinite computation because the data variables of type `Nat` can grow indefinitely. The finite approach also can not instantiate the data variables of type `Nat` and therefore a PBES is returned. The signature for each data variable when using the finite approach without enumerations is:

```
X :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
Y :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
```

When using enumerations, the signature becomes:

```
X :: Nat x Nat x Nat x Nat -> Bool
Y :: Nat x Nat x Nat x Nat -> Bool
```

The property can not be checked using BES-solvers, because the result is a PBES and not a BES.

BAK4 The μ-calculus formula which describes that a process inevitably enters a critical section, is the following:

```
mu X. [!enter(true)]X
```

The PBES for BAK4 can be found in appendix D.8 and consist of one parameterised boolean equation with predicate variable X. The signature of X is the following:

```
X :: Pos x Nat x Bool x Nat x Pos x Nat x Bool x Nat -> Bool
```

The PBES using enumerations have the following signature:

```
X :: Enum6 x Nat x Bool x Nat x Enum6 x Nat x Bool x Nat -> Bool
```

The number of equations computed for each approach, together with the time needed to compute the result is shown in the following table.

Type	**lazy**	**lazy enum**	**finite**	**finite enum**
# equations	37	37	4	144
Time	0.25	0.19	0.10	0.29

Table 6.8: Results for BAK4

The lazy approach leads to a finite BES with 37 equations, while the statespace of the system is of infinite size. The BES is finite, because all paths starting from an `enter(true)` action do not have to be considered, because `enter(true)` has occurred.
The finite approach can not instantiate the data variables of type `Nat` and therefore a PBES is returned. The signature for each data variable when using the finite approach without enumerations is:

```
X :: Pos x Nat x Nat x Pos x Nat x Nat -> Bool
```

When using enumerations, the signature becomes:

```
X :: Nat x Nat x Nat x Nat -> Bool
```

The property holds on Lamport's bakery protocol.

Conclusion Lamport's Bakery Protocol shows that there are specifications where the lazy approach will have an infinite computation. The finite approach however is guaranteed to terminate, resulting in a (smaller) PBES, which can make symbolic model checking easier. The fourth property however, shows that it is possible that in a system with an infinite state space, a finite BES can be computed with the lazy approach.

Chapter 7

Conclusions

To verify if a property holds on a system, it suffices to compute the solution of the PBES, which represents the combination of the system and property. A step towards computing the solution of PBESs, is instantiating them. This instantiation process can reduce the complexity of the equations in the PBES, eventually leading to a BES. In this thesis we have considered two approaches for instantiating PBESs.

The finite approach instantiates all data variables of finite data type. Using the finite approach, the result will not necessarily be a BES, but the right hand sides of the equations in the PBES can be reduced in complexity by instantiating all finite data types. Because no data variables of infinite data type are instantiated, the computation is guaranteed to terminate. The finite approach can increase the performance of symbolic approximation techniques due to the reduced complexity of the equations.

The lazy approach assumes the PBES to be solved in the initial state. Therefore it only computes those equations the initial state depends on. In most cases, this will lead to a smaller BES than the finite approach would lead to (if it would result in a BES at all). If the initial state depends on an infinite number of equations, the lazy approach will result in an infinite computation. On systems with infinite state spaces, it can be the case that a property on such a system leads to a finite BES in the lazy approach. For example the fourth property we considered on Lamport's bakery protocol leads to a finite BES.

The case studies have shown that (using BES-solvers) it is possible to verify a number of properties on systems by instantiating PBESs. The alternating bit protocol (which has a finite state space) has shown that the lazy approach can lead to (much) smaller BESs then the finite approach could lead to. Lamport's bakery protocol is a system with an infinite state space and for a lot of properties, the lazy approach would result in an infinite computation. Therefore the finite approach can be used in those cases, to reduce the complexity of the PBES. For some properties on Lamport's bakery protocol the lazy approach will lead to a finite BES.

As we have seen, it is now possible with the *mCRL2-toolset* to verify properties on systems, although external BES-solvers must be used, because a BES-solver is not yet present in the toolset.

7.1 Future work

The performance of the finite and lazy approach depends heavily on the complexity of the predicate formulae in the PBES. Therefore, the performance can be improved greatly, if predicate formulae can be dealt with in a more effective way. By implementing a combination of substitution and rewriting, in a lot of cases (big) parts of predicate formulae does not have to be computed.
Another optimisation which can be implemented is the improved sorting algorithm, which will reduce the time used by sorting the result in the lazy approach.

It is possible to rewrite predicate formulae by implementing the quantifier elimination techniques. In the lazy approach, this will lead to BESs for a bigger subset of PBESs and both in the finite and lazy approach, the complexity of the resulting PBESs and BESs will be reduced.

The function Find_Dummy to get a random instantiation for a data variable (as used by the instantiation of free variables) can be useful in more tools on PBESs. Therefore it is a good idea to implement this function as part of the PBES-library.

The CADP-format as output format when a BES is computed could be implemented to be able to solve BESs using the *CADP-toolset*.

Bibliography

[BaW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[BeG94] M.A. Bezem and J.F. Groote. *Invariants in Process Algebra with Data.* In B. Jonsson and J. Parrow, editors, *Proceedings 5th Conference on Concurrency Theory (CONCUR'94)*, Lecture Notes in Computer Science, volume 836, pages 401-416. Springer-Verlag, 1994.

[CADP] http://www.inrialpes.fr/vasy/cadp/.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. *The Concurrency Workbench: A semantics-based tool for the verification of finite-state systems.* In ACM Transactions on Programming Languages and Systems, volume 15(1), pages 36-72. ACM Press, January 1993.

[DPW07] A. van Dam, S.C.W. Ploeger and T.A.C. Willemse. *Instantiation for Parameterised Boolean Equation Systems.* in preparation, 2007.

[Gar05] H. Garavel. *The Open/Caesar Reference Manual.* Chapter 16, pages 143-165. 2005.

[Gro97] J.F. Groote. *The syntax and semantics of timed μCRL.* Technical Report SEN-R9709. CWI, 1997

[GrP94] J.F. Groote and A. Ponse. *The syntax and semantics of μCRL.* In Algebra of Communicating Processes, Workshops in Computing, A. Ponse et al., pages 26-62. 1994.

[GrM99] J.F. Groote and R. Mateescu. *Verification of Temporal Properties of Processes in a Setting with Data.* In A.M. Haeberer, editor, *Algebraic Methodology and Software Technology: 7th International Conference (AMAST'98)*, Lecture Notes in Computer Science, volume 1548, pages 74-90. Springer-Verlag, 1999.

[GrR01] J.F. Groote and M. Reniers. *Algebraic process verification.* In Handbook of Process Algebra, J.A. Bergstra et al., pages 1151-1208. Elsevier Science, 2001.

[GMP06] J.F. Groote, A.H.J. Mathijssen, S.C.W. Ploeger, M.A. Reniers, M.J. van Weerdenburg and J. van der Wulp. *Process Algebra and mCRL2.* IPA Basic Course on Formal Methods 2006. January 2006.

[GMR07] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko and M.J. van Weerdenburg. *The Formal Specification Language mCRL2.* To appear in: Proc. Methods for Modelling Software Systems. Dagstuhl Seminar Proceedings 06351 (2007).

[GrW05] J.F. Groote and T.A.C. Willemse. *Parameterised Boolean Equation Systems.* In S. Abramsky and M. Mavronicolas, editors, Theoretical Computer Science, volume 343, pages 332-369. Elsevier, 2005.

[GrW05a] J.F. Groote and T.A.C. Willemse. *Model-checking processes with data.* In Science of Computer Programming, volume 56, pages 251-273. Elsevier, 2005.

[Koz83] D. Kozen. *Results on the propositional μ-calculus.* In M. Nivat, editor, Theoretical Computer Science, volume 27, pages 333-354. Elsevier, 1983

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems.* PhD thesis. Technical University of Munich, 1997.

[Mat03] R. Mateescu. *A Generic On-the-Fly Solver for Alternation-Free Boolean Equation Systems.* In Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2003 (Warsaw, Poland). April 2003

[McM92] K.L. McMillan. *Symbolic Model Checking.* PhD thesis. Carnegie Mellon University, 1992

[MCRL] http://homepages.cwi.nl/ mcrl/.

[MCRL2] http://www.mcrl2.org/.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion.* North Oxford Academic, 1986.

Appendix A

Formats

A.1 Internal format of PBESs

This chapter describes the grammar of PBESs in terms of the internal format.

PBES

`<PBES> ::= PBES(<DataSpec>, <PBEqn>*, <PropVarInst>)`
In the internal format a PBES takes a data specification, which contains rewrite rules, definitions and declarations on the data. Also it takes zero or more parameterised boolean equations and an initial state, which is empty if the PBES has no parameterised boolean equations.

Predicate Variable Instantiation

`<PropVarInst> ::= PropVarInst(<String>, <DataExpr>*)`
A `<PropVarInst>` is the internal representation of a predicate variable instantiation.

PBES equation

`<PBEqn> ::= PBEqn(<FixPoint>, <PropVarDecl>, <PBExpr>)`
A parameterised boolean equation has a fixpoint, a predicate variable and a parameterised boolean expression which represents the right hand side of the parameterised boolean equation.

Fixpoint symbol

```
<FixPoint>  ::= Mu
            |   Nu
```

Predicate Variable

`<PropVarDecl> ::= PropVarDecl(<String>, <DataVarId>*)`
A `<PropVarDecl>` is the internal representation of a predicate variable.

PBES expression

```
<PBExpr>    ::= <DataExpr>
            |   PBESTrue
            |   PBESFalse
            |   PBESAnd(<PBExpr>, <PBExpr>)
            |   PBESOr(<PBExpr>, <PBExpr>)
            |   PBESForall(<DataVarId>+, <PBExpr>)
            |   PBESExists(<DataVarId>+, <PBExpr>)
            |   <PropVarInst>
```

where a `<DataExpr>` contains an arbitrary data expression and a `<DataVarId>` is an arbitrary data variable.

A.2 CWI format of a BES

CWI uses a representation of a BES, which is a straightforward derivation of the BESs as described in section 4.1.

Representation of operators, constants

Math-representation	CWI-representation
μ	`min`
ν	`max`
$\wedge$	&
$\vee$	|
$\top$	`T`
$\bot$	`F`

Grammar of the CWI representation of a BES

The grammar for a BES in the CWI format, is the following:

```
<axiom> ::= <bes>
<bes> ::= <boolean_equation*>
<boolean_equation> ::= <fixpoint> <predicate_variable> = <bes_expression>
<fixpoint> ::= min
      | max
<predicate_variable> ::= <natural_number>
<bes_expression> ::= F
      | T
      | ( <bes_expression> & <bes_expression> )
      | ( <bes_expression> | <bes_expression> )
      | <predicate_variable>
```

Note that:

- `<natural-number>` is a non-negative integer.

Example of a BES in the CWI format

```
max 0 = 1 & 2
max 1 = 0 | 1 | 2
max 2 = 5 & 3
max 3 = 1 | 4
max 4 = T
min 5 = 6 | 7
min 6 = F
min 7 = 7 & 8
min 8 = 5 | 6 | 8
```

This represents the following BES:
$\nu X_0 = X_1 \wedge X_2$
$\nu X_1 = X_0 \vee X_1 \vee X_2$
$\nu X_2 = Y_0 \wedge X_3$
$\nu X_3 = X_1 \vee X_4$
$\nu X_4 = \top$
$\mu Y_0 = Y_1 \vee Y_2$
$\mu Y_1 = \perp$
$\mu Y_2 = Y_2 \wedge Y_3$
$\mu Y_3 = Y_0 \vee Y_1 \vee Y_3$

A.3 CADP format of a BES

In the CADP format, a BES is a sequence of boolean equation blocks, in which each boolean equation block consists of boolean equations with the same fixpoint symbol.

Representation of operators, constants

Math-representation	**CADP-representation**
μ	`mu`
ν	`nu`
$\wedge$	`and`
$\vee$	`or`
$\top$	`true`
$\bot$	`false`

Requirements on boolean equation blocks:

- For a BES with N boolean equation blocks, each boolean equation block has an index in the range [0...N).
- Every boolean equation block has a number of boolean variables, called Xi, where i is the index, starting at 0.
- Each boolean equation block is identified by a blockname Bi, where i is the index of the boolean equation block.
- Every boolean equation block has a fixpoint (`mu`, `nu`) assigned.
- At the left hand side of a boolean equation only one predicate variable is allowed, and it has to be a unique variable for that boolean equation block.
- At the right hand side of a boolean equation only predicate variables, constants (`true`, `false`) and operators (`and`, `or`) are allowed.
- A local variable is a variable in the right hand side of a boolean equation which is declared in the boolean equation block it appears.
- A global variable is a variable in the right hand side of a boolean equation which is declared in another boolean equation block than it appears.
- Every predicate variable must be declared somewhere in the boolean equation block or in another boolean equation block if it is a global variable.
- Every boolean equation has only one type of operators. Boolean equations with only `and` operators are called conjunctive, boolean equations with only `or` operators are called disjunctive.
- The empty disjunctive equation is `false`, the empty conjunctive operator is `true`.
- Xi depends on Xj if Xj is used in the right hand side of boolean equation Xi.
- A boolean equation block Bi depends on Block Bj if a variable in Bi depends on a variable in Bj.

Order

The definition on the order of parameterised boolean equations in PBESs (definition 4.5) can be extended to boolean equation blocks, by taking boolean equation blocks for the parameterised boolean equations and the complete BES for the PBES.

The order of boolean equations in a boolean equation block may be changed, because they have the same fixpoint symbol.

Grammar of the CADP representation of a BES

The grammar for the BES is a slightly adapted variant from the grammar presented in [Gar05]. To allow empty BESs, an addition `<empty-bes>` is made to `<block-list>`, denoting an empty sequence of boolean equation blocks. The `<unique>` and `<mode>` clauses are omitted, because they are implementation specific for the CADP-toolkit, and thus not interesting for representing BESs The grammar for a BES, as in [Gar05], is the following:

```
<axiom> ::= <block-list>
<block-list> ::= <block>
         | <block> <block-list>
         | <empty-bes>
<block> ::= block <sign> <block-identifier> is
          <equation-list>
       end block
<sign> ::= mu
      | nu
<block-identifier> ::= B<natural-number>
<equation-list> ::= <equation>
      | <equation> <equation-list>
<equation> ::= <local-variable-identifier>
<local-variable-identifier> ::= X<natural-number>
<global-variable-identifier> ::= X<natural-number>_<natural-number>
<formula> ::= <atomic-formula>
      | <disjunctive-formula>
      | <conjunctive-formula>
<atomic-formula> ::= false
      | true
      | <local-variable-identifier>
      | <global-variable-identifier>
<disjunctive-formula> ::= <atomic-formula> or <atomic-formula>
      | <atomic-formula> or <disjunctive-formula>
<conjunctive-formula> ::= <atomic-formula> and <atomic-formula>
      | <atomic-formula> and <conjunctive-formula>
```

Note that:

- `<natural-number>` is a non-negative integer.
- `X<natural-number>` denotes a local variable with respect to the current block. So `X1` is variable `X1` in the current block. Each local variable can be written in a global form, by adding the identifier of the block in the variable name.
- `X<natural-number>_<natural-number>` denotes a global variable. So `X0_1` is variable `X0` in block `B1`.
- Blocks must appear by an ordering of increasing blocknumbers.

Example of a BES in the CADP format

```
block nu B0 is
    X0 = X1 and X2
    X1 = X0 or X1 or X2
    X2 = X0_1 and X3
    X3 = X1 or X4
    X4 = true
end block

block mu B1 is
    X0 = X1 or X2
    X1 = false
    X2 = X2 and X3
    X3 = X0 or X1 or X3
end block
```

A possible representation of the above BES in the format used in the previous chapter (using different predicate variables for different fixpoint operators):
$\nu X_0 = X_1 \wedge X_2$
$\nu X_1 = X_0 \vee X_1 \vee X_2$
$\nu X_2 = Y_0 \wedge X_3$
$\nu X_3 = X_1 \vee X_4$
$\nu X_4 = \top$
$\mu Y_0 = Y_1 \vee Y_2$
$\mu Y_1 = \perp$
$\mu Y_2 = Y_2 \wedge Y_3$
$\mu Y_3 = Y_0 \vee Y_1 \vee Y_3$

Appendix B

Tools

In this appendix we give a short overview of the tools which are written as result of the research.

B.1 Pbesinfo

To gain insight in the used PBES-format in the *mCRL2-toolset*, a tool named *pbesinfo* has been made. This tool has the same usage as many other tools in the toolkit. This means that an input file is given to the tool (say *input.pbes*). To read the input-file, the *LPS-framework* is used. The tool *pbesinfo* will show information about the PBES.

Information displayed

Pbesinfo displays the following information:

- If the PBES is well formed and / or closed
- Number of parameterised boolean equations
- Number of μ's
- Number of ν's
- The set of predicate variables which have a μ as a fixpoint
- The set of predicate variables which have a ν as a fixpoint
- All predicate variables, with their signature.

The last three shown parts are optional (using commandline option -f/–full), because in big systems, this leads to unreadable output.

Example B.1. Assume the following PBES (the representation of the PBES in the example is not the actual representation of a PBES in binary form):

$$\begin{aligned}
\mu X(n{:}\mathbb{N}) &= Y(1/3 * n, n+1) \vee n > 10 \\
\nu Y(r{:}\mathbb{R}, n{:}\mathbb{N}) &= Z(\bot, r, n-1) \wedge X(n) \\
\mu Z(b{:}\mathbb{B}, r{:}\mathbb{R}, n{:}\mathbb{N}) &= (b == \bot \wedge X(n+1)) \vee (b == \top \wedge Y(r, n))
\end{aligned}$$

Pbesinfo will show the following information:

```
The PBES is closed and well formed
Number of equations:   3
Number of mu's:        2   (X, Z)
Number of nu's:        1   (Y)
Predicate variables:   X ::  Nat -> Bool
                       Y ::  Real x Nat -> Bool
                       Z ::  Bool x Real x Nat -> Bool
```

B.2 Pbes2bes

The tool *pbes2bes* implements the finite and lazy approach as defined in chapter 5. The tool takes a PBES and depending if a name for an outputfile is given, it will put the result to the console or to the file specified.

The tool can be run with the following tool-specific options:

- -s/–strategy. Use the specified strategy (lazy or finite).
- -o/–output. Use the specified output format (binary, internal or CWI).

Appendix C

Example lazy approach

In this appendix we show how the transformation for the lazy approach is done step by step.

Take a PBES:
$\mu X(b{:}\mathbb{B}) = Y(\bot) \wedge X(b)$
$\nu Y(b{:}\mathbb{B}) = X(b)$
with initial state $X(\top)$.
We denote $\mu X(b{:}\mathbb{B}) = Y(\bot) \wedge X(b)$ by σ_1 and $\nu Y(b{:}\mathbb{B}) = X(b)$ by σ_2, so we denote the whole PBES as $\sigma_1\sigma_2$.

```
    trans( $X(\top), \sigma_1\sigma_2$ )
=               {Definition trans }
    sort( trans′( $\{X(\top)\}, \emptyset, \sigma_1\sigma_2$ ), $\sigma_1\sigma_2$ )
=               {Definition trans′}
    sort(
        pbecreate($X(\top), \sigma_1\sigma_2$)
        trans′( $(\emptyset \cup \mathsf{dep}(X(\top), \sigma_1\sigma_2)) \backslash (\emptyset \cup \{X(\top)\}), \emptyset \cup \{X(\top)\}, \sigma_1\sigma_2$ )
    , $\sigma_1\sigma_2$)
=               {Definition pbecreate, see sub1; dep, see sub2; algebra}
    sort(
        $\mu X_\top = Y_\bot \wedge X_\top$
        trans′( $\{Y(\bot), X(\top)\} \backslash \{X(\top)\}, \{X(\top)\}, \sigma_1\sigma_2$ )
    , $\sigma_1\sigma_2$)
=               {Algebra}
    sort(
        $\mu X_\top = Y_\bot \wedge X_\top$
        trans′( $\{Y(\bot)\}, \{X(\top)\}, \sigma_1\sigma_2$ )
    , $\sigma_1\sigma_2$)
=               {Definition trans′}
    sort(
        $\mu X_\top = Y_\bot \wedge X_\top$
        pbecreate($Y(\bot), \sigma_1\sigma_2$)
        trans′( $(\emptyset \cup \mathsf{dep}(Y(\bot), \sigma_1\sigma_2)) \backslash (\{X(\top)\} \cup \{Y(\bot)\}), \{X(\top)\} \cup \{Y(\bot)\}, \sigma_1\sigma_2$ )
    , $\sigma_1\sigma_2$)
=               {Definition pbecreate, see sub3; dep, see sub4; algebra}
    sort(
        $\mu X_\top = Y_\bot \wedge X_\top$
        $\nu Y_\bot = X_\bot$
        trans′( $\{X(\bot)\} \backslash \{X(\top), Y(\bot)\}, \{X(\top), Y(\bot)\}, \sigma_1\sigma_2$ )
    , $\sigma_1\sigma_2$)
```

$=$ $\{\text{Algebra}\}$
sort(
$\mu X_\top = Y_\bot \wedge X_\top$
$\nu Y_\bot = X_\bot$
trans′($\{X(\bot)\}, \{X(\top), Y(\bot)\}, \sigma_1\sigma_2$)
$, \sigma_1\sigma_2)$

$=$ $\{\text{Definition } \mathsf{trans'}\}$
sort(
$\mu X_\top = Y_\bot \wedge X_\top$
$\nu Y_\bot = X_\bot$
pbecreate$(X(\bot), \sigma_1\sigma_2)$
trans′($(\emptyset \cup \mathsf{dep}(X(\top), \sigma_1\sigma_2))\backslash(\{X(\top), Y(\bot)\} \cup \{X(\top)\}), \{X(\top), Y(\bot)\} \cup \{X(\top)\}, \sigma_1\sigma_2$)
$, \sigma_1\sigma_2)$

$=$ {Definition pbecreate, see sub5; dep, see sub6; algebra}
sort(
$\mu X_\top = Y_\bot \wedge X_\top$
$\nu Y_\bot = X_\bot$
$\mu X_\bot = Y_\bot \wedge X_\bot$
trans′($\{Y(\bot), X(\bot)\}\backslash(\{X(\top), Y(\bot), X(\top)\}), \{X(\top), Y(\bot), X(\top)\}, \sigma_1\sigma_2$)
$, \sigma_1\sigma_2)$

$=$ $\{\text{Algebra}\}$
sort(
$\mu X_\top = Y_\bot \wedge X_\top$
$\nu Y_\bot = X_\bot$
$\mu X_\bot = Y_\bot \wedge X_\bot$
trans′($\emptyset, \{X(\top), Y(\bot), X(\bot)\}, \sigma_1\sigma_2$)
$, \sigma_1\sigma_2)$

$=$ {Definition trans′}
sort($(\mu X_\top = Y_\bot \wedge X_\top)$ $(\nu Y_\bot = X_\bot)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$, $\sigma_1\sigma_2)$

$=$ {Definition sort }
$(\mu X_\top = Y_\bot \wedge X_\top)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$
sort($(\mu X_\top = Y_\bot \wedge X_\top)$ $(\nu Y_\bot = X_\bot)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$, $\sigma_2)$

$=$ {Definition sort }
$(\mu X_\top = Y_\bot \wedge X_\top)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$ $(\nu Y_\bot = X_\bot)$
sort($(\mu X_\top = Y_\bot \wedge X_\top)$ $(\nu Y_\bot = X_\bot)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$, $\epsilon)$

$=$ {Definition sort }
$(\mu X_\top = Y_\bot \wedge X_\top)$ $(\mu X_\bot = Y_\bot \wedge X_\bot)$ $(\nu Y_\bot = X_\bot)$

Sub1 pbecreate$(X(\top), \sigma_1\sigma_2)$

pbecreate$(X(\top), \sigma_1\sigma_2)$

$=$ {Definition pbecreate }
$\mu X_\top = \mathsf{subst}(Y(\bot) \wedge X(\top), \mathcal{X})$

$=$ {Definition subst }
$\mu X_\top = \mathsf{subst}(Y(\bot), \mathcal{X}) \wedge \mathsf{subst}(X(\top), \mathcal{X})$

$=$ {Definition subst }
$\mu X_\top = Y_\bot \wedge X_\top$

Sub2 $\mathsf{dep}(X(\top), \sigma_1\sigma_2)$

$\mathsf{dep}(X(\top), \sigma_1\sigma_2)$
$=$ $\{$Definition dep $\}$
$\mathsf{instset}(Y(\bot) \wedge X(\top))$
$=$ $\{$Definition $\mathsf{instset}$ $\}$
$\mathsf{instset}(Y(\bot)) \cup \mathsf{instset}(X(\top))$
$=$ $\{$Definition $\mathsf{instset}$ $\}$
$\{Y(\bot)\} \cup \{X(\top)\}$
$=$ $\{$Algebra$\}$
$\{Y(\bot), X(\top)\}$

Sub3 $\mathsf{pbecreate}(Y(\bot), \sigma_1\sigma_2)$

$\mathsf{pbecreate}(Y(\bot), \sigma_1\sigma_2)$
$=$ $\{$Definition $\mathsf{pbecreate}$ $\}$
$\mathsf{pbecreate}(Y(\bot), \sigma_2)$
$=$ $\{$Definition $\mathsf{pbecreate}$ $\}$
$\nu Y_\bot = \mathsf{subst}(X(\bot), \mathcal{X})$
$=$ $\{$Definition subst $\}$
$\nu Y_\bot = X_\bot$

Sub4 $\mathsf{dep}(Y(\bot), \sigma_1\sigma_2)$

$\mathsf{dep}(Y(\bot), \sigma_1\sigma_2)$
$=$ $\{$Definition dep $\}$
$\mathsf{dep}(Y(\bot), \sigma_2)$
$=$ $\{$Definition dep $\}$
$\mathsf{instset}(X(\bot))$
$=$ $\{$Definition $\mathsf{instset}$ $\}$
$\{X(\bot)\}$

Sub5 $\mathsf{pbecreate}(X(\bot), \sigma_1\sigma_2)$

$\mathsf{pbecreate}(X(\bot), \sigma_1\sigma_2)$
$=$ $\{$Definition $\mathsf{pbecreate}$ $\}$
$\mu X_\bot = \mathsf{subst}(Y(\bot) \wedge X(\bot), \mathcal{X})$
$=$ $\{$Definition subst $\}$
$\mu X_\bot = \mathsf{subst}(Y(\bot), \mathcal{X}) \wedge \mathsf{subst}(X(\bot), \mathcal{X})$
$=$ $\{$Definition subst $\}$
$\mu X_\bot = Y_\bot \wedge X_\bot$

Sub6 $\mathsf{dep}(X(\bot), \sigma_1\sigma_2)$

$\mathsf{dep}(X(\bot), \sigma_1\sigma_2)$
$=$ $\{$Definition dep $\}$
$\mathsf{instset}(Y(\bot) \wedge X(\bot))$
$=$ $\{$Definition $\mathsf{instset}$ $\}$
$\mathsf{instset}(Y(\bot)) \cup \mathsf{instset}(X(\bot))$
$=$ $\{$Definition $\mathsf{instset}$ $\}$
$\{Y(\bot)\} \cup \{X(\bot)\}$
$=$ $\{$Algebra$\}$
$\{Y(\bot), X(\bot)\}$

Appendix D

PBESs of alternating bit protocol and Lamport's bakery protocol

In this appendix we show the PBESs which resulted from the case study.

D.1 ABP1

```
nu X(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool) =
  (((((((((
    (exists d0_00: D. val(s30 == 1))
  ||
    val(s31 == 4 && s33 == 1))
  ||
    (exists e5_00: Bool. val((s31 == 3 && s33 == 1)
            && if(e5_00, b2, !b2) == b4)))
  ||
    (exists e3_00: Bool. val(s32 == 2)))
  ||
    val(s33 == 2))
  ||
    (exists e4_00: Bool. val(s32 == 1 && if(e4_00, s33 == 4, s33 == 3))))
  ||
    (exists e2_00: Bool. val(s31 == 2)))
  ||
    (exists e0_00: Bool. val((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3)))
  ||
    val(s30 == 2 && s31 == 1))
  ||
    val(s30 == 3 && s32 == 4))
&&
  ((((((((
    (forall d0_00: D. val(!(s30 == 1))
            || X(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2))
  &&
    val(!(s31 == 4 && s33 == 1))
            || X(s30, d, b, 1, freevar7, freevar8, s32, b3, 4, freevar15, b2))
  &&
    (forall e5_00: Bool. val(!((s31 == 3 && s33 == 1)
  &&
    if(e5_00, b2, !b2) == b4))
            || X(s30, d, b, 1, freevar5, freevar6, s32, b3,
                 if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2)))
  &&
    (forall e3_00: Bool. val(!(s32 == 2))
            || X(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                 if(e3_00, freevar10, b3), s33, d6, b2)))
  &&
    val(!(s33 == 2))
            || X(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2))
  &&
    (forall e4_00: Bool. val(!(s32 == 1 && if(e4_00, s33 == 4, s33 == 3)))
            || X(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                 C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2))))
  &&
    (forall e2_00: Bool. val(!(s31 == 2))
            || X(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                 if(e2_00, freevar4, b4), s32, b3, s33, d6, b2)))
  &&
    (forall e0_00: Bool. val(!((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3))
            || X(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0),
                 if(e0_00, b, !b), s31, d7, b4, 1, freevar11, s33, d6, b2)))
  &&
    val(!(s30 == 2 && s31 == 1))
            || X(3, d, b, 2, d, b, s32, b3, s33, d6, b2))
&&
  val(!(s30 == 3 && s32 == 4))
||
  X(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2);

init X(1, freevar, true, 1, freevar1, freevar2,
       1, freevar9, 1, freevar13, true);
```

D.2 ABP2

```
nu X(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool) =
  ((((((((
    (forall d0_00: D. val(!(s30 == 1))
            || X(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2))
  &&
    val(!(s31 == 4 && s33 == 1))
            || X(s30, d, b, 1, freevar7, freevar8, s32, b3, 4, freevar15, b2))
  &&
    (forall e5_00: Bool. val(!((s31 == 3 && s33 == 1)
            && if(e5_00, b2, !b2) == b4))
            || X(s30, d, b, 1, freevar5, freevar6, s32, b3,
                 if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2)))
  &&
    (forall e3_00: Bool. val(!(s32 == 2))
            || X(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                 if(e3_00, freevar10, b3), s33, d6, b2)))
  &&
    val(!(s33 == 2))
            || X(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2))
  &&
    (forall e4_00: Bool. val(!(s32 == 1 && if(e4_00, s33 == 4, s33 == 3)))
            || X(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                 C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2))))
  &&
    (forall e2_00: Bool. val(!(s31 == 2))
            || X(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                 if(e2_00, freevar4, b4), s32, b3, s33, d6, b2)))
  &&
    (forall e0_00: Bool. val(!((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3))
            || X(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0),
                 if(e0_00, b, !b), s31, d7, b4, 1, freevar11, s33, d6, b2)))
  &&
    val(!(s30 == 2 && s31 == 1))
            || X(3, d, b, 2, d, b, s32, b3, s33, d6, b2))
  &&
    val(!(s30 == 3 && s32 == 4))
            || X(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2))
&&
  (exists d00: D. (((((forall d0_00: D. (val(d0_00 != d00)
                || val(!(s30 == 1)))
                || Y(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2, d00))
      && (forall e5_00: Bool. true))
      && (forall e3_00: Bool. true))
      && (forall e4_00: Bool. true))
      && (forall e2_00: Bool. true))
      && (forall e0_00: Bool. true));

nu Y(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool, d00: D) =
  ((((((((
    (exists d0_00: D. val(s30 == 1)
            && Y(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2, d00))
  ||
    val(s31 == 4 && s33 == 1)
            && Y(s30, d, b, 1, freevar7, freevar8, s32,
                 b3, 4, freevar15, b2, d00))
  ||
    (exists e5_00: Bool. val((s31 == 3 && s33 == 1)
            && if(e5_00, b2, !b2) == b4)
            && Y(s30, d, b, 1, freevar5, freevar6, s32, b3,
                 if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2, d00)))
  ||
    (exists e3_00: Bool. val(s32 == 2)
            && Y(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                 if(e3_00, freevar10, b3), s33, d6, b2, d00)))
  ||
    val(s33 == 2)
            && Y(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2, d00))
  ||
    (exists e4_00: Bool. val(s32 == 1 && if(e4_00, s33 == 4, s33 == 3))
            && Y(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                 C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2), d00)))
  ||
    (exists e2_00: Bool. val(s31 == 2)
             && Y(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                  if(e2_00, freevar4, b4), s32, b3, s33, d6, b2, d00)))
  ||
    (exists e0_00: Bool. val((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3)
            && Y(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0), if(e0_00, b, !b),
                 s31, d7, b4, 1, freevar11, s33, d6, b2, d00)))
  ||
    val(s30 == 2 && s31 == 1)
            && Y(3, d, b, 2, d, b, s32, b3, s33, d6, b2, d00))
  ||
    val(s30 == 3 && s32 == 4)
            && Y(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2, d00))
||
  (((((
    (exists d0_00: D. false)
  ||
    (exists e5_00: Bool. false))
  ||
    (exists e3_00: Bool. false))
  ||
    val(d6 == d00) && val(s33 == 2))
  ||
    (exists e4_00: Bool. false))
  ||
    (exists e2_00: Bool. false))
||  (exists e0_00: Bool. false);

init X(1, freevar, true, 1, freevar1, freevar2,
       1, freevar9, 1, freevar13, true);
```

D.3 ABP3

```
nu X(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
          s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool) =
  ((((((((
    (forall d0_00: D. val(!(s30 == 1))
            || X(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2))
  &&
    val(!(s31 == 4 && s33 == 1))
            || X(s30, d, b, 1, freevar7, freevar8, s32, b3, 4, freevar15, b2))
  &&
    (forall e5_00: Bool. val(!((s31 == 3 && s33 == 1)
            && if(e5_00, b2, !b2) == b4))
            || X(s30, d, b, 1, freevar5, freevar6, s32, b3,
                 if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2)))
  &&
    (forall e3_00: Bool. val(!(s32 == 2))
            || X(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                 if(e3_00, freevar10, b3), s33, d6, b2)))
  &&
    val(!(s33 == 2))
            || X(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2))
  &&
    (forall e4_00: Bool. val(!(s32 == 1 && if(e4_00, s33 == 4, s33 == 3)))
            || X(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                 C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2))))
  &&
    (forall e2_00: Bool. val(!(s31 == 2))
            || X(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                 if(e2_00, freevar4, b4), s32, b3, s33, d6, b2)))
  &&
    (forall e0_00: Bool. val(!((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3))
            || X(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0),
                 if(e0_00, b, !b), s31, d7, b4, 1, freevar11, s33, d6, b2)))
  &&
    val(!(s30 == 2 && s31 == 1))
            || X(3, d, b, 2, d, b, s32, b3, s33, d6, b2))
  &&
    val(!(s30 == 3 && s32 == 4))
            || X(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2))
&&
  (exists dd: D. (((((forall d0_00: D. (val(d0_00 != dd)
                             || val(!(s30 == 1)))
                             || Y(2, d0_00, b, s31, d7, b4, s32, b3,
                                  s33, d6, b2, dd))
                             && (forall e5_00: Bool. true))
                             && (forall e3_00: Bool. true))
                             && (forall e4_00: Bool. true))
          && (forall e2_00: Bool. true))
          && (forall e0_00: Bool. true));

nu Y(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool, dd: D) =
  Z(s30, d, b, s31, d7, b4, s32, b3, s33, d6, b2, dd);

mu Z(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool, dd: D) =
  ((((((((
    (forall d0_00: D. val(!(s30 == 1))
            || Z(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2, dd))
  &&
    val(!(s31 == 4 && s33 == 1))
            || Z(s30, d, b, 1, freevar7, freevar8, s32,
                 b3, 4, freevar15, b2, dd))
  &&
    (forall e5_00: Bool. val(!((s31 == 3 && s33 == 1)
            && if(e5_00, b2, !b2) == b4))
            || Z(s30, d, b, 1, freevar5, freevar6, s32, b3,
               if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2, dd)))
  &&
    (forall e3_00: Bool. true))
  &&
    (val(d6 == dd)
            || val(!(s33 == 2)))
            || Z(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2, dd))
  &&
    (forall e4_00: Bool. val(!(s32 == 1 && if(e4_00, s33 == 4, s33 == 3)))
            || Z(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                 C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2), dd)))
  &&
    (forall e2_00: Bool. true))
  &&
    (forall e0_00: Bool. val(!((s30 == 3 && s32 == 3)
            && if(e0_00, !b, b) == b3))
            || Z(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0), if(e0_00, b, !b),
                 s31, d7, b4, 1, freevar11, s33, d6, b2, dd)))
  &&
    val(!(s30 == 2 && s31 == 1))
            || Z(3, d, b, 2, d, b, s32, b3, s33, d6, b2, dd))
  &&
    val(!(s30 == 3 && s32 == 4))
            || Z(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2, dd))
 &&
((((
    (forall d0_00: D. true)
  &&
    (forall e5_00: Bool. true))
  &&
    (forall e3_00: Bool. val(!(s32 == 2))
            || Y(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                 if(e3_00, freevar10, b3), s33, d6, b2, dd)))
  &&
    (forall e4_00: Bool. true))
  &&
    (forall e2_00: Bool. val(!(s31 == 2))
             || Y(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                  if(e2_00, freevar4, b4), s32, b3, s33, d6, b2, dd)))
```

```
&&
    (forall e0_00: Bool. true);

init X(1, freevar, true, 1, freevar1, freevar2,
       1, freevar9, 1, freevar13, true);
```

D.4 ABP4

```
nu X(s30: Pos, d: D, b: Bool, s31: Pos, d7: D, b4: Bool,
     s32: Pos, b3: Bool, s33: Pos, d6: D, b2: Bool) =
  exists dd: D. ((((((((
        (forall d0_00: D. (val(d0_00 == dd) || val(!(s30 == 1)))
              || X(2, d0_00, b, s31, d7, b4, s32, b3, s33, d6, b2))
      &&
        val(!(s31 == 4 && s33 == 1))
              || X(s30, d, b, 1, freevar7, freevar8,
                   s32, b3, 4, freevar15, b2))
      &&
        (forall e5_00: Bool. val(!((s31 == 3 && s33 == 1)
              && if(e5_00, b2, !b2) == b4))
              || X(s30, d, b, 1, freevar5, freevar6, s32, b3,
                   if(e5_00, 2, 4), C2_fun(e5_00, d7, freevar14), b2)))
    &&
      (forall e3_00: Bool. val(!(s32 == 2))
              || X(s30, d, b, s31, d7, b4, if(e3_00, 4, 3),
                   if(e3_00, freevar10, b3), s33, d6, b2)))
    &&
      val(!(s33 == 2))
              || X(s30, d, b, s31, d7, b4, s32, b3, 3, freevar16, b2))
    &&
      (forall e4_00: Bool. val(!(s32 == 1 && if(e4_00, s33 == 4, s33 == 3)))
              || X(s30, d, b, s31, d7, b4, 2, if(e4_00, !b2, b2), 1,
                   C2_fun(e4_00, freevar18, freevar17), if(e4_00, b2, !b2))))
    &&
      (forall e2_00: Bool. val(!(s31 == 2))
              || X(s30, d, b, if(e2_00, 4, 3), C2_fun(e2_00, freevar3, d7),
                   if(e2_00, freevar4, b4), s32, b3, s33, d6, b2)))
    &&
      (forall e0_00: Bool. val(!((s30 == 3 && s32 == 3)
              && if(e0_00, !b, b) == b3))
              || X(if(e0_00, 2, 1), C2_fun(e0_00, d, freevar0),
                   if(e0_00, b, !b), s31, d7, b4, 1, freevar11, s33, d6, b2)))
    &&
      val(!(s30 == 2 && s31 == 1))
              || X(3, d, b, 2, d, b, s32, b3, s33, d6, b2))
              && val(!(s30 == 3 && s32 == 4))
              || X(2, d, b, s31, d7, b4, 1, freevar12, s33, d6, b2))
&&
  (((((
    (forall d0_00: D. true)
  &&
    (forall e5_00: Bool. true))
  &&
    (forall e3_00: Bool. true))
  &&
    val(d6 != dd) || val(!(s33 == 2)))
  &&
    (forall e4_00: Bool. true))
  &&
    (forall e2_00: Bool. true))
```

```
&&
  (forall e0_00: Bool. true);

init X(1, freevar, true, 1, freevar1, freevar2,
       1, freevar9, 1, freevar13, true);
```

D.5 BAK1

```
 nu X(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos, m3: Nat, b0: Bool, n0: Nat) =
  ((((((
    (val(s3 == 5) || val(s3 == 1))
  ||
    (exists e3_00: Bool. val(if(e3_00, s3 == 6, s3 == 4 && n < m
                            || m == 0))))
  ||
    (exists e9_00: Bool. val(if(e9_00, s30 == 6, s30 == 4 && n0 < m3
                            || m3 == 0))))
  ||
    val(s30 == 1))
  ||
    val(s30 == 5))
  ||
    (exists e4_00: Enum3, e8_00: Enum7.
      val((C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
      && C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
      && !b == b0)))
  ||
    (exists e_00: Enum7, e10_00: Enum3.
      val((C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
                  s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
      && C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
      && !b0 == b)))
&&
  ((((((
    (val(!(s3 == 5))
     || X(1, freevar10, b, freevar11, s30, m3, b0, n0))
  &&
    val(!(s3 == 1))
     || X(2, freevar1, b, 0, s30, m3, b0, n0))
  &&
    (forall e3_00: Bool. val(!if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))
            || X(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0)))
  &&
    (forall e9_00: Bool.
            val(!if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))
             || X(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0)))
  &&
    val(!(s30 == 1))
     || X(s3, m, b, n, 2, freevar18, b0, 0))
  &&
    val(!(s30 == 5))
     || X(s3, m, b, n, 1, freevar27, b0, freevar28))
  &&
    (forall e4_00: Enum3, e8_00: Enum7.
      val(!((C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
      && C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
      && !b == b0))
    ||
      X(C3_fun1(e4_00, 3, 4, 4),
        C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
        b,
        C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
        C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
        C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25,
                freevar23, freevar22, freevar19),
        b0,
        C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20))))
&&
  (forall e_00: Enum7, e10_00: Enum3.
      val(!(
       (C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
               s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1) &&
                  C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0
                  && !(m3 == 0)))
      && !b0 == b))
    ||
      X(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
        C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8,
                freevar6, freevar5, freevar2),
        b,
        C7_fun0(e_00, n, n, n, n, n, n, freevar3),
        C3_fun1(e10_00, 3, 4, 4),
        C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                C7_fun0(e_00, n, n, n, n, n, n, 0)),
        b0,
        C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0)));

init X(1, freevar, true, freevar0, 1, freevar16, false, freevar17);
```

D.6 BAK2

```
nu X(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos, m3: Nat, b0: Bool, n0: Nat) =
  (((((((
    (val(!(s3 == 5)) || X(1, freevar10, b, freevar11, s30, m3, b0, n0))
  &&
    val(!(s3 == 1)) || X(2, freevar1, b, 0, s30, m3, b0, n0))
  &&
    (forall e3_00: Bool. val(!if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))
            || X(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0)))
  &&
    (forall e9_00: Bool. val(!if(e9_00,
                                s30 == 6,
                                s30 == 4 && n0 < m3 || m3 == 0))
            || X(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0)))
  &&
    val(!(s30 == 1))
            || X(s3, m, b, n, 2, freevar18, b0, 0)) && val(!(s30 == 5))
            || X(s3, m, b, n, 1, freevar27, b0, freevar28))
  &&
    (forall e4_00: Enum3, e8_00: Enum7.
        val(!((
         C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
         &&
          C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                 s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
         &&
          !b == b0))
      ||
        X(C3_fun1(e4_00, 3, 4, 4),
          C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                  C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
          b,
          C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
          C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25,
                  freevar23, freevar22, freevar19),
          b0,
          C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20))))
  &&
    (forall e_00: Enum7, e10_00: Enum3.
        val(!((
         C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
                s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
         &&
         C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
         && !b0 == b))
      ||
        X(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8,
                  freevar6, freevar5, freevar2),
          b,
          C7_fun0(e_00, n, n, n, n, n, n, freevar3),
          C3_fun1(e10_00, 3, 4, 4),
          C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                  C7_fun0(e_00, n, n, n, n, n, n, 0)),
          b0,
          C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0))))
&&
  (exists b00: Bool.
     (((((
       (val(b != b00)
     ||
        val(!(s3 == 1)))
     ||
        Y(2, freevar1, b, 0, s30, m3, b0, n0, b00))
     &&
        (forall e3_00: Bool. true))
     &&
        (forall e9_00: Bool. true))
     &&
        (val(b0 != b00) || val(!(s30 == 1)))
           || Y(s3, m, b, n, 2, freevar18, b0, 0, b00))
     &&
        (forall e4_00: Enum3, e8_00: Enum7. true))
     &&
        (forall e_00: Enum7, e10_00: Enum3. true));

mu Y(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos,
     m3: Nat, b0: Bool, n0: Nat, b00: Bool) =
  (((((((
    (val(s3 == 5) && Y(1, freevar10, b, freevar11, s30, m3, b0, n0, b00))
  ||
    val(s3 == 1) && Y(2, freevar1, b, 0, s30, m3, b0, n0, b00))
  ||
    (exists e3_00: Bool.
       val(if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))
        && Y(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0, b00)))
  ||
    (exists e9_00: Bool.
       val(if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))
        && Y(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0, b00)))
  ||
    val(s30 == 1) && Y(s3, m, b, n, 2, freevar18, b0, 0, b00))
  ||
    val(s30 == 5) && Y(s3, m, b, n, 1, freevar27, b0, freevar28, b00))
  ||
    (exists e4_00: Enum3, e8_00: Enum7.
        val((
         C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
        &&
         C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
        &&
         !b == b0)
      &&
        Y(C3_fun1(e4_00, 3, 4, 4),
          C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
        b,
        C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
        C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
        C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25,
                freevar23, freevar22, freevar19),
        b0,
        C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20), b00)))
  ||
    (exists e_00: Enum7, e10_00: Enum3.
       val((
        C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
               s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
       &&
        C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
       &&
        !b0 == b)
     &&
       Y(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
        C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8,
                freevar6, freevar5, freevar2),
        b,
        C7_fun0(e_00, n, n, n, n, n, n, freevar3),
        C3_fun1(e10_00, 3, 4, 4),
        C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                C7_fun0(e_00, n, n, n, n, n, n, 0)),
        b0,
        C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0),
        b00)))
||
  (((exists e3_00: Bool.
          val(b == b00)
       &&
          val(if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0)))
       ||
          (exists e9_00: Bool. val(b0 == b00)
                 && val(if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))))
       ||
          (exists e4_00: Enum3, e8_00: Enum7. false))
       ||
          (exists e_00: Enum7, e10_00: Enum3. false);

init X(1, freevar, true, freevar0, 1, freevar16, false, freevar17);
```

D.7 BAK3

```
nu X(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos, m3: Nat, b0: Bool, n0: Nat) =
  (((((((
    (val(!(s3 == 5)) || X(1, freevar10, b, freevar11, s30, m3, b0, n0))
  &&
    val(!(s3 == 1)) || X(2, freevar1, b, 0, s30, m3, b0, n0))
  &&
    (forall e3_00: Bool.
            val(!if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))
            || X(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0)))
  &&
    (forall e9_00: Bool.
            val(!if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))
            || X(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0)))
  &&
    val(!(s30 == 1)) || X(s3, m, b, n, 2, freevar18, b0, 0))
  &&
    val(!(s30 == 5)) || X(s3, m, b, n, 1, freevar27, b0, freevar28))
  &&
    (forall e4_00: Enum3, e8_00: Enum7.
        val(!((
         C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
        &&
         C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
        &&
            !b == b0))
      ||
        X(C3_fun1(e4_00, 3, 4, 4),
          C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                  C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
          b,
          C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
          C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25, freevar23,
                  freevar22, freevar19),
          b0,
          C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20))))
  &&
    (forall e_00: Enum7, e10_00: Enum3.
        val(!((
         C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
                s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
        &&
         C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
        &&
         !b0 == b))
      ||
        X(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8, freevar6,
                  freevar5, freevar2),
          b,
          C7_fun0(e_00, n, n, n, n, n, n, freevar3),
          C3_fun1(e10_00, 3, 4, 4),
          C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                  C7_fun0(e_00, n, n, n, n, n, n, 0)),
          b0,
          C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0))))
&&
  (exists b00: Bool.
     (((((
        (val(b != b00) || val(!(s3 == 1)))
       ||
        Y(2, freevar1, b, 0, s30, m3, b0, n0, b00))
       &&
        (forall e3_00: Bool. true))
       &&
        (forall e9_00: Bool. true))
       &&
        (val(b0 != b00) || val(!(s30 == 1)))
       ||
        Y(s3, m, b, n, 2, freevar18, b0, 0, b00))
       &&
        (forall e4_00: Enum3, e8_00: Enum7. true))
       &&
        (forall e_00: Enum7, e10_00: Enum3. true));


mu Y(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos, m3: Nat,
     b0: Bool, n0: Nat, b00: Bool) =
  ((((((((
    (val(!(s3 == 5)) || Y(1, freevar10, b, freevar11, s30, m3, b0, n0, b00))
  &&
    val(!(s3 == 1)) || Y(2, freevar1, b, 0, s30, m3, b0, n0, b00))
  &&
    (forall e3_00: Bool.
          val(!if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))
          || Y(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0, b00)))
  &&
    (forall e9_00: Bool.
          val(!if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))
          || Y(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0, b00)))
  &&
    val(!(s30 == 1)) || Y(s3, m, b, n, 2, freevar18, b0, 0, b00))
  &&
    val(!(s30 == 5)) || Y(s3, m, b, n, 1, freevar27, b0, freevar28, b00))
  &&
    (forall e4_00: Enum3, e8_00: Enum7.
        val(!((
         C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
        &&
         C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
               s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
        &&
         !b == b0))
       ||
        Y(C3_fun1(e4_00, 3, 4, 4),
          C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                  C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
          b,
          C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
          C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25, freevar23,
                  freevar22, freevar19),
          b0,
          C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20),
          b00)))
  &&
    (forall e_00: Enum7, e10_00: Enum3.
        val(!((
         C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
               s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
        &&
         C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
        &&
         !b0 == b))
       ||
        Y(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8, freevar6,
                  freevar5, freevar2),
          b,
          C7_fun0(e_00, n, n, n, n, n, n, freevar3),
          C3_fun1(e10_00, 3, 4, 4),
          C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                  C7_fun0(e_00, n, n, n, n, n, n, 0)),
          b0,
          C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0),
          b00)))
  &&
    (((((
       (val(s3 == 5) || val(s3 == 1))
    ||
       (exists e3_00: Bool. val(if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0))))
    ||
       (exists e9_00: Bool.
             val(if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))))
    ||
       val(s30 == 1))
    ||
       val(s30 == 5))
    ||
       (exists e4_00: Enum3, e8_00: Enum7.
          val((
           C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
          &&
           C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                  s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
          &&
           !b == b0)))
  ||
    (exists e_00: Enum7, e10_00: Enum3.
        val((
         C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
                s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
        &&
         C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
        &&
         !b0 == b)))
||
    (((exists e3_00: Bool. val(b == b00)
              && val(if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0)))
  ||
    (exists e9_00: Bool. val(b0 == b00)
            && val(if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0))))
  ||
    (exists e4_00: Enum3, e8_00: Enum7. false))
||
  (exists e_00: Enum7, e10_00: Enum3. false);

init X(1, freevar, true, freevar0, 1, freevar16, false, freevar17);
```

D.8 BAK4

```
nu X(s3: Pos, m: Nat, b: Bool, n: Nat, s30: Pos, m3: Nat, b0: Bool, n0: Nat) =
  ((((((
    (val(!(s3 == 5)) || X(1, freevar10, b, freevar11, s30, m3, b0, n0))
  &&
    val(!(s3 == 1)) || X(2, freevar1, b, 0, s30, m3, b0, n0))
  &&
    (forall e3_00: Bool.
            (val(b == true)
          ||
            val(!if(e3_00, s3 == 6, s3 == 4 && n < m || m == 0)))
          ||
            X(5, if(e3_00, freevar13, freevar7), b, n, s30, m3, b0, n0)))
  &&
    (forall e9_00: Bool.
            (val(b0 == true)
          ||
            val(!if(e9_00, s30 == 6, s30 == 4 && n0 < m3 || m3 == 0)))
          ||
            X(s3, m, b, n, 5, if(e9_00, freevar30, freevar24), b0, n0)))
  &&
    val(!(s30 == 1)) || X(s3, m, b, n, 2, freevar18, b0, 0))
  &&
    val(!(s30 == 5)) || X(s3, m, b, n, 1, freevar27, b0, freevar28))
  &&
    (forall e4_00: Enum3, e8_00: Enum7.
          val(!((C3_fun(e4_00, s3 == 2, s3 == 3, s3 == 4 && m <= n && !(m == 0))
        &&
          C7_fun(e8_00, s30 == 6, s30 == 5, s30 == 4 && m3 <= n0 && !(m3 == 0),
                 s30 == 4 && n0 < m3 || m3 == 0, s30 == 3, s30 == 2, s30 == 1))
        &&
          !b == b0))
      ||
        X(C3_fun1(e4_00, 3, 4, 4),
          C3_fun0(e4_00, freevar4, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0),
                  C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0)),
          b,
          C3_fun0(e4_00, C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, 0) + 1, n, n),
          C7_fun1(e8_00, 6, 5, 3, 6, 3, 2, 1),
          C7_fun0(e8_00, freevar31, freevar29, freevar26, freevar25,
                  freevar23, freevar22, freevar19),
          b0,
          C7_fun0(e8_00, n0, n0, n0, n0, n0, n0, freevar20))))
&&
  (forall e_00: Enum7, e10_00: Enum3.
         val(!((C7_fun(e_00, s3 == 6, s3 == 5, s3 == 4 && m <= n && !(m == 0),
                     s3 == 4 && n < m || m == 0, s3 == 3, s3 == 2, s3 == 1)
       &&
         C3_fun(e10_00, s30 == 2, s30 == 3, s30 == 4 && m3 <= n0 && !(m3 == 0)))
       &&
         !b0 == b))
     ||
       X(C7_fun1(e_00, 6, 5, 3, 6, 3, 2, 1),
         C7_fun0(e_00, freevar14, freevar12, freevar9, freevar8,
                 freevar6, freevar5, freevar2),
         b,
         C7_fun0(e_00, n, n, n, n, n, n, freevar3),
         C3_fun1(e10_00, 3, 4, 4),
         C3_fun0(e10_00, freevar21, C7_fun0(e_00, n, n, n, n, n, n, 0),
                 C7_fun0(e_00, n, n, n, n, n, n, 0)),
         b0,
         C3_fun0(e10_00, C7_fun0(e_00, n, n, n, n, n, n, 0) + 1, n0, n0)));

init X(1, freevar, true, freevar0, 1, freevar16, false, freevar17);
```