EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

A symbolic approach to PBES instantiation

Boshoven, T.P.M.

Award date:
2015

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

A Symbolic Approach to PBES Instantiation

Tom Boshoven, BSc
March 13, 2013

Master's Thesis
Computer Science and Engineering

Supervisor:
T.A.C. Willemse, PhD

Eindhoven University of Technology
Department of Mathematics and Computer Science

Abstract

We give a complete approach for instantiation of parametrized boolean equation systems (PBESs) and the subsequent solving of the resulting boolean equation systems. The given approach makes use of techniques used in symbolic state-space explorations. In this approach, we first transform the PBES into a special form named clustered PBES, and subsequently instantiate this clustered PBES into a BES, represented by a structure graph, using symbolic techniques. We present algorithms for solving this BES, making use of symbolic data structures.

Contents

1 Introduction **3**
1.1 Overview . . . 4

2 Background **5**
2.1 Boolean Equation Systems . . . 5
2.1.1 Structure Graphs . . . 7
2.2 Parametrized Boolean Equation Systems . . . 12
2.2.1 Explicit Instantiation . . . 14

3 Symbolic Instantiation **15**
3.1 Motivation . . . 15
3.2 Approach . . . 15
3.3 State-space . . . 16
3.4 Clusters . . . 17
3.5 Guarded Normal Form . . . 19
3.5.1 Strongly Guarded Form . . . 20
3.5.2 Transformation . . . 22
3.5.3 Correctness of the Transformation . . . 24
3.6 Clustered GNF . . . 32

4 Exploration **35**
4.1 Next-state function . . . 35
4.2 Explicit Exploration . . . 36
4.3 Symbolic Exploration . . . 36
4.3.1 Partitioned Transition Relation . . . 36
4.4 Event Locality . . . 38
4.4.1 Dependency . . . 38
4.4.2 Local transition relation . . . 40
4.5 Merging Groups . . . 41
4.6 Linked Decision Diagrams . . . 42
4.6.1 Basic Operations . . . 44
4.6.2 States and Transitions . . . 47
4.7 Algorithm . . . 49
4.7.1 Usage . . . 51
4.8 BESsyness . . . 52
4.9 Optimizations . . . 53
4.9.1 True/false-elimination . . . 53

5 Solving **58**
5.1 Parity Games . . . 58
5.1.1 Simple Recursive Form . . . 58
5.2 Transformation to Parity Games . . . 59
5.3 Directly . . . 60
5.3.1 Recursive Algorithm . . . 60

5.3.2 Gauß Elimination 61

6 Conclusion 65

7 Future Work 66

A Coq 69

A.1 PBES Framework 69

A.2 Strongly Guarded Form 71

1 Introduction

Formal verification of safety-critical systems is often desired because of the severe consequences of failure in such systems. Model checking is a technique for performing validation of requirements on systems. It is successfully used to find flaws in various algorithms and systems ([30, 17]).

The model checking problem is the problem of checking whether a model of a system conforms to the requirements given to that system. This model is usually an abstracted software or hardware system, and the requirements are generally temporal. In [25], it is shown that a given model checking problem can be translated into a system of boolean fixed-point equations, or Boolean Equation System (BES). Solving such a system corresponds to finding the solution to the represented model checking problem.

Because experience showed that BESs are not sufficiently versatile in settings in which data plays a role, the BES formalism was extended with data in [15], resulting in parametrized boolean equation systems (PBESs).

In addition to the model checking problem, PBESs are useful for equivalence checking using various bisimulations ([9]) and for static analysis of code ([13]).

A general approach for solving a PBES first eliminates the parameters using a process called explicit instantiation. This process results in a boolean equation system, which can be solved algorithmically. This approach is sketched in Figure 1.

Figure 1: A common approach to PBES instantiation.

Instantiation of a PBES can be seen as a type of state-space exploration. In [20], Gijs Kant and Jaco van de Pol present an approach for instantiation of PBESs into parity games, a formalism which specializes BESs. This approach uses existing state-space exploration techniques in order to perform the instantiation.

We present a generalized approach to this type of instantiation of PBESs, in which the result of the instantiation is a BES. This approach uses symbolic state-space exploration techniques for the instantiation. Furthermore, optimizations of the resulting BES are discussed and algorithms for solving this BES are presented. The objective of this work is providing a starting point for integration of a symbolic PBES instantiation technique into the mCRL2 toolset[1].

It was shown in [20] that in some cases, using symbolic methods in the instantiation of PBESs vastly reduces the time and memory cost of the process. This makes it possible to solve model-checking problems that can not be solved using an approach based on explicit instantiation in reasonable time. Because of this, it is very useful to have a tool which performs instantiation using symbolic exploration methods. The approach used in [20] has some problems and limitations, which we identify and overcome.

One of the identified problems occurs in a preprocessing step, which transforms equations in a type of conjunctive normal form. In some cases, performing this step exponentially blows up a system, or even results into a system which can no longer be instantiated. An example of such a system, along with a more detailed explanation, is given in Section 3.1. This problem is solved by weakening the required form, and defining a new transformation to this form, which does not suffer from this behavior.

Furthermore, the current approach for symbolic instantiation of PBESs instantiates into a parity game. Parity games are a formalism which corresponds to a strict subset of BESs. We generalize the approach by instantiating a PBESs into BESs. This reduces the difference in structure between

[1] `http://mcrl2.org/`

the original PBES and the instantiated system, which may help improve tracing of verification results.

1.1 Overview

First, the background concepts are explained and used definitions are given in Section 2. Over the following three sections, we present a complete approach for finding the solution to a given PBES. This approach is sketched in Figure 2.

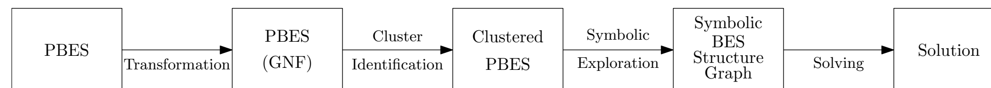


Figure 2: A symbolic approach to PBES instantiation.

The approach consists of three general steps. Section 3 describes the requirements for the symbolic exploration approach. In addition, it provides the necessary transformations for PBESs in order to fulfill these requirements. This is done by means of a special form (Strongly Guarded Form or SGF) and identification of subformulas called clusters. Following this, Section 4 details the process of symbolic exploration of a PBES. This results in a symbolic representation of a state-space corresponding to the resulting boolean equation system. Lastly, Section 5 contains algorithms for solving this symbolic BES.

2 Background

2.1 Boolean Equation Systems

A model-checking problem can be represented as a boolean equation system (BES) by combining a model with its desired properties into a single system ([31]). Finding the *solution* to a BES corresponds to answering the represented model-checking problem.

Definition 2.1 (BES) *A boolean equation system is a system of boolean fixed-point equations, characterized by the following grammar:*

$$\begin{aligned} BES &::= \epsilon \\ &\mid (\sigma\mathcal{X} = \Phi)\ BES \\ \Phi &::= \mathcal{X} \mid (\Phi \wedge \Phi) \mid (\Phi \vee \Phi) \mid true \mid false \end{aligned}$$

Here, ϵ is the empty system, σ is either μ or ν and $\mathcal{X}$ represents any variable. The right-hand sides of the equations in a BES (denoted by Φ) are called proposition formulas.

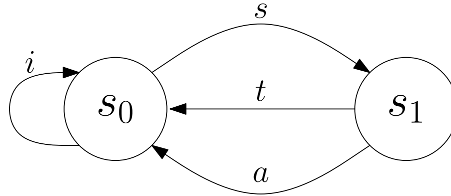
For a BES $\mathcal{E}$, we define $occ(\mathcal{E})$ (occurring variables) as the set of recursion variables occurring in the right-hand side of at least one equation in $\mathcal{E}$ and we define $bnd(\mathcal{E})$ (bound variables) as the set of recursion variables occurring in the left-hand side of an equation.

Note that we do not allow negations to occur in the equations. All equations in the system are preceded by a ν (greatest) or μ (least) fixed-point symbol. In a BES, the ordering of the equations is of importance for the solution to the system.

We assume *closed* BESs. This means that all variables occurring in a right-hand side of an equation (Φ) must be in the left-hand side of exactly one equation ($occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$).

In the remainder of this document, we often use an alternative syntax in which equations are placed on separate lines and excess brackets are removed. This notation can be seen in Example 1.

Example 1 (BES) *We model a simple process $\mathbb{P}$, described by the following labeled transition system:*



Process $\mathbb{P}$ may perform internal actions i, or send a message using action s. After sending a message, it either receives an acknowledgment (action a) or encounters a timeout (action t).

We can describe the property that there exists a path through $\mathbb{P}$ in which an s action is done infinitely often, but an a action can be done only finitely often (a message is sent infinitely often, but can be acknowledged only finitely often). Intuitively, this property is false *when starting from either state of $\mathbb{P}$, because after each s step, an a step can always be done.*

A possible way to represent this property in the modal μ calculus ([28]) is the following formula f:

$$f = \mu X.(\nu Y.(\mu Z.(\langle s\rangle\, Y \vee \langle\overline{s \vee a}\rangle\, Z) \wedge [a]X))$$

The verification of f on process $\mathbb{P}$ is a model checking problem. By combining f with $\mathbb{P}$, a boolean

equation system corresponding to this problem can be created:

$$\begin{aligned}
\mu X_0 &= Y_0\\
\mu X_1 &= Y_1\\
\nu Y_0 &= Z_0\\
\nu Y_1 &= Z_1\\
\mu Z_0 &= (Y_1 \vee Z_0) \wedge true\\
\mu Z_1 &= (false \vee Z_0) \wedge X_0
\end{aligned}$$

We have $\mathbb{P}, s_j \models f$ (f holds in state s_j of $\mathbb{P}$, with $j \in \{0,1\}$) if and only if X_j has the solution true in this BES.

The semantics for BESs are derived from the semantics of fixed-point equation systems (of which BESs are a specialization) which are defined in [25]. In [21], BES semantics are defined as follows:

Definition 2.2 (BES semantics) *Let $\eta : \mathbf{X} \to \mathbb{B}$ be an environment mapping recursion variables $\mathcal{X} \in \mathbf{X}$ to a boolean value. The* interpretation *$[\![f]\!]\eta$ maps a proposition formula f to true or $false$:*

$$\begin{aligned}
[\![true]\!]\eta =& true\\
[\![false]\!]\eta =& false\\
[\![\mathcal{X}]\!]\eta =& \eta(\mathcal{X})\\
[\![\Phi_1 \vee \Phi_2]\!]\eta =& [\![\Phi_1]\!]\eta \vee [\![\Phi_2]\!]\eta\\
[\![\Phi_1 \wedge \Phi_2]\!]\eta =& [\![\Phi_1]\!]\eta \wedge [\![\Phi_2]\!]\eta
\end{aligned}$$

The solution of a BES given environment η is inductively defined as follows:

$$\begin{aligned}
[\![\epsilon]\!]\eta =& \eta\\
[\![(\mu\mathcal{X} = \Phi)\mathcal{E}]\!]\eta =& [\![\mathcal{E}]\!]\eta[\mathcal{X} := [\![\Phi]\!]([\![\mathcal{E}]\!]\eta[\mathcal{X} := false])]\\
[\![(\nu\mathcal{X} = \Phi)\mathcal{E}]\!]\eta =& [\![\mathcal{E}]\!]\eta[\mathcal{X} := [\![\Phi]\!]([\![\mathcal{E}]\!]\eta[\mathcal{X} := true])]
\end{aligned}$$

Here, the expression $\eta[X := b]$ assigns value b to $\eta(X)$.

Solving a BES $\mathcal{E}$ for recursion variable $\mathcal{X} \in \mathbf{X}$ means finding the solution of $([\![\mathcal{E}]\!]\eta)(\mathcal{X})$ for some environment η.

Example 2 (BES solution) *Let $\mathcal{E}$ be the BES from Example 1.*

Calculating $[\![\mathcal{E}]\!]\eta$ for some environment η yields some environment η', such that:

$$\begin{aligned}
\eta'(X_0) =& false\\
\eta'(X_1) =& false\\
\eta'(Y_0) =& false\\
\eta'(Y_1) =& false\\
\eta'(Z_0) =& false\\
\eta'(Z_1) =& false
\end{aligned}$$

We omit the calculation resulting in η', because of the length of the intermediate expressions.

By observing that in this environment, X_0 and X_1 are both $false$, we can conclude that property f from Example 1 does indeed not hold from s_0 or s_1.

Various algorithms exist to solve BESs efficiently. Some of these algorithms, such as the recursive algorithm by Zielonka ([32]), the small progress measures algorithm by Jurdziński ([19]), and the model checking algorithm by Stevens and Stirling ([27]) require transformation of the system to a normal form in order to solve them. Other algorithms, such as Gauß elimination ([24]) can solve BESs directly.

2.1.1 Structure Graphs

In Section 3.3, we show that in order to efficiently use a symbolic data structure for representing a boolean equation system, it is desirable to represent such a system as a graph. One of the ways to do this is by means of a structure graph for boolean equation systems, introduced by Jeroen Keiren, Michel Reniers and Tim Willemse in [22].

Definition 2.3 (Structure graph) *We define a structure graph $\mathcal{G}$ as a 5-tuple $\langle S, s_0, \rightarrow, d, r\rangle$ where:*

- *S is a finite set of nodes*
- *s_0 is the initial node*
- *$\rightarrow \subseteq S \times S$ is a dependency relation*
- *$d : S \mapsto \{\wedge, \vee, \top, \bot\}$ is a node decoration mapping*
- *$r : S \mapsto \mathbb{N}$ is a node ranking mapping*

Here, d and r are partial functions.

In our definition, we diverge from [22], where a mapping $\nearrow$ is included. This mapping assigns free variables to nodes. Because we only consider BESs in which there are no free variables in the scope of this document, we disregard this element, implicitly giving it a value of $\emptyset$.

In a structure graph constructed from a BES, each node corresponds to a subformula of this BES. The dependency relation $\rightarrow$ describes occurrence of a subformula in a formula, or occurrence of a recursion variable in a formula. For example, the formula $\nu X = Y \vee (X \wedge Z)$ corresponds to a node describing X. This node has an outgoing edge to the node describing Y and another one to the node corresponding to $(X \wedge Z)$. This last node has an outgoing edge to the node describing Z and an outgoing edge back to the node describing X. A complete example is given in Example 4.

The node decoration mapping d describes whether the subformula described by the node is a constant ($true$ or $false$) or a conjunction or disjunction at the highest level of their parse tree. The nodes corresponding to $true$ get decoration $\top$ and the nodes corresponding to $false$ get decoration $\bot$. Conjunctions are decorated with the $\wedge$ symbol and disjunctions have decoration $\vee$. For instance, a node describing $X \vee Y$ is decorated with the $\vee$ symbol. For a node s with a single dependency Z, the value of $d(s)$ may be left undefined.

The ranking mapping r attaches a natural number, named *rank*, to the nodes corresponding to fixed-point equations (in contrast to subformulas of these equations). The rank of such a node defines whether the equation to which it corresponds has a greatest or least fixed-point, and also determines the ordering of equations. If a node s does not correspond to an equation, the value of $r(s)$ is undefined.

Definition 2.4 (Rank) *Assume a boolean equation system $\mathcal{E}$, consisting of N equations of the form $(\sigma_i \mathcal{X}_i = \Phi_i)$ (with $1 \leqslant i \leqslant N$), where σ_i is a fixed-point symbol. The rank $rank_{\mathcal{E}}(\mathcal{X}_i)$ of a single recursion variable $\mathcal{X}_i$ in the context of system $\mathcal{E}$ is determined as follows.*

$$rank_{\mathcal{E}}(\mathcal{X}_i) = \begin{cases} 0 & \textit{if } i = N \wedge \sigma_i = \nu \\ 1 & \textit{if } i = N \wedge \sigma_i = \mu \\ rank_{\mathcal{E}}(\mathcal{X}_{i+1}) & \textit{if } i < N \wedge \sigma_i = \sigma_{i+1} \\ rank_{\mathcal{E}}(\mathcal{X}_{i+1}) + 1 & \textit{if } i < N \wedge \sigma_i \neq \sigma_{i+1} \end{cases} \tag{1}$$

Example 3 (Rank) *Assume the BES $\mathcal{E}$ from Example 1:*

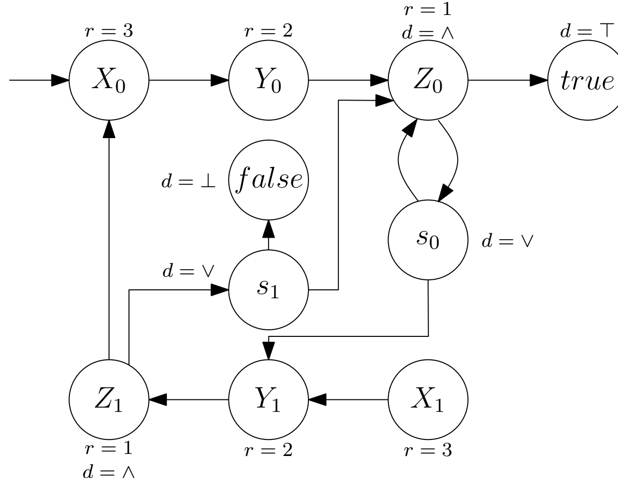
$$\begin{aligned} \mu X_0 &= Y_0 \\ \mu X_1 &= Y_1 \\ \nu Y_0 &= Z_0 \\ \nu Y_1 &= Z_1 \\ \mu Z_0 &= (Y_1 \vee Z_0) \wedge true \\ \mu Z_1 &= (false \vee Z_0) \wedge X_0 \end{aligned}$$

The rank of the variables occurring in the BES can be determined as follows:

$$\begin{array}{rll} rank_{\mathcal{E}}(Z_1) & & = 1 \\ rank_{\mathcal{E}}(Z_0) & = rank_{\mathcal{E}}(Z_1) & = 1 \\ rank_{\mathcal{E}}(Y_1) & = rank_{\mathcal{E}}(Z_0) + 1 & = 2 \\ rank_{\mathcal{E}}(Y_0) & = rank_{\mathcal{E}}(Y_1) & = 2 \\ rank_{\mathcal{E}}(X_1) & = rank_{\mathcal{E}}(Y_0) + 1 & = 3 \\ rank_{\mathcal{E}}(X_0) & = rank_{\mathcal{E}}(X_1) & = 3 \end{array}$$

Example 4 (Structure graph) *Assume the BES from Examples 1 and 3.*

A structure graph representation of this BES is given below:



Here, the nodes corresponding to recursion variables are labeled with the name of this variable. The nodes corresponding to constants true and $false$ are labeled as such. The node labeled s_0 corresponds to the subformula $(Y_1 \vee Z_0)$ and the node labeled s_1 corresponds to the subformula $(false \vee Z_0)$. The states are annotated with their corresponding values of r and d, where defined. The initial node is X_0, as we want to solve $\mathcal{E}$ for X_0.

Note that nodes have a rank if and only if they correspond to a recursion variable and a decoration if their number of outgoing edges is unequal to 1.

In [22], structured operational semantics for deriving structure graphs from BES are presented. We restrict these semantics to structure graphs corresponding to closed BESs ($occ(\mathcal{E}) \subseteq bnd(\mathcal{E})$ for equation system $\mathcal{E}$).

Using these SOS rules, it is possible to construct a structure graph corresponding to a given boolean equation system, as done manually in Example 4.

Definition 2.5 (Structured operational semantics for structure graphs) *The following SOS-rules describe the relation between a structure graph and a formula f in the context of an equation system $\mathcal{E}$, denoted as $\langle f, \mathcal{E} \rangle$.*

$$\frac{\mathcal{X} \in occ(\mathcal{E})}{r(\langle \mathcal{X}, \mathcal{E} \rangle) = rank_{\mathcal{E}}(\mathcal{X})} \quad (1)$$

$$\frac{}{d(\langle true, \mathcal{E} \rangle) = \top} \quad (2) \qquad \frac{}{d(\langle false, \mathcal{E} \rangle) = \bot} \quad (3)$$

$$\frac{}{d(\langle f \wedge g, \mathcal{E} \rangle) = \wedge} \quad (4) \qquad \frac{}{d(\langle f \vee g, \mathcal{E} \rangle) = \vee} \quad (5)$$

$$\frac{d(\langle f, \mathcal{E} \rangle) = \wedge \qquad \langle f, \mathcal{E} \rangle \notin dom(r) \qquad \langle f, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle} \quad (6)$$

$$\frac{d(\langle g, \mathcal{E} \rangle) = \wedge \qquad \langle g, \mathcal{E} \rangle \notin dom(r) \qquad \langle g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle} \quad (7)$$

$$\frac{d(\langle f, \mathcal{E} \rangle) = \vee \qquad \langle f, \mathcal{E} \rangle \notin dom(r) \qquad \langle f, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle} \quad (8)$$

$$\frac{d(\langle g, \mathcal{E} \rangle) = \vee \qquad \langle g, \mathcal{E} \rangle \notin dom(r) \qquad \langle g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle h, \mathcal{E} \rangle} \quad (9)$$

$$\frac{d(\langle f, \mathcal{E} \rangle) \neq \wedge}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (10) \qquad \frac{d(\langle g, \mathcal{E} \rangle) \neq \wedge}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (11)$$

$$\frac{d(\langle f, \mathcal{E} \rangle) \neq \vee}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (12) \qquad \frac{d(\langle g, \mathcal{E} \rangle) \neq \vee}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (13)$$

$$\frac{\langle f, \mathcal{E} \rangle \in dom(r)}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (14) \qquad \frac{\langle f, \mathcal{E} \rangle \in dom(r)}{\langle f \wedge g, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (15)$$

$$\frac{\langle g, \mathcal{E} \rangle \in dom(r)}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (16) \qquad \frac{\langle g, \mathcal{E} \rangle \in dom(r)}{\langle f \vee g, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (17)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad d(\langle f, \mathcal{E} \rangle) = \wedge \qquad \langle f, \mathcal{E} \rangle \notin dom(r)}{d(\langle \mathcal{X}, \mathcal{E} \rangle) = \wedge} \quad (18)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad d(\langle f, \mathcal{E} \rangle) = \vee \qquad \langle f, \mathcal{E} \rangle \notin dom(r)}{d(\langle \mathcal{X}, \mathcal{E} \rangle) = \vee} \quad (19)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad \langle f, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle \qquad d(\langle f, \mathcal{E} \rangle) = \wedge \qquad \langle f, \mathcal{E} \rangle \notin dom(r)}{\langle \mathcal{X}, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (20)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad \langle f, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle \qquad d(\langle f, \mathcal{E} \rangle) = \vee \qquad \langle f, \mathcal{E} \rangle \notin dom(r)}{\langle \mathcal{X}, \mathcal{E} \rangle \rightarrow \langle g, \mathcal{E} \rangle} \quad (21)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad d(\langle f, \mathcal{E} \rangle) \notin \{\vee, \wedge\}}{\langle \mathcal{X}, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (22)$$

$$\frac{(\sigma\mathcal{X} = f) \in \mathcal{E} \qquad \langle f, \mathcal{E} \rangle \in dom(r)}{\langle \mathcal{X}, \mathcal{E} \rangle \rightarrow \langle f, \mathcal{E} \rangle} \quad (23)$$

Rule (1) describes the fact that all nodes corresponding to recursion variables are ranked. Rules (2) to (5) describe the value of the decoration d. Rules (6) to (9) flatten the nesting hierarchy for unranked nodes with the same decoration. They can be used to deduce that $\langle X \wedge (Y \wedge Z), \mathcal{E} \rangle \rightarrow \langle Y, \mathcal{E} \rangle$. This flattening is only allowed within the representation of a single equation. Rules (10) to

(13) handle the cases that flattening is not possible, due to switches in decoration. These rules can be used to deduce that $\langle X \wedge (Y \vee Z), \mathcal{E}\rangle \rightarrow \langle Y \vee Z, \mathcal{E}\rangle$, but do not allow the deduction that $\langle X \wedge (Y \wedge Z), \mathcal{E}\rangle \rightarrow \langle Y \wedge Z, \mathcal{E}\rangle$, forcing the structure to be flattened to a single node. Rules (14) to (17) define the other case where flattening as in rules (6) to (1) is not allowed, namely when the inner node is ranked. This can be used to deduce that $\langle X \wedge Y, \mathcal{E}\rangle \rightarrow \langle Y, \mathcal{E}\rangle$ if $d(Y) = \wedge$ and Y is a recursion variable. Rules (18) to (23) describe how the structure is derived for a recursion variable. Rules (18) to (21) describe the case where the right-hand side of the equation is a conjunction or disjunction. Rules (20) and (21) describe the cases where the right-hand side is a constant or variable.

A structure graph can be constructed from any boolean equation system, using the rules in Definition 2.5. However, not every structure graph represents a sane BES. Therefore, when constructing a BES from a structure graph, we require mild restrictions on the structure graph. These restrictions are collected in the *BESsyness* property ([22]).

Definition 2.6 (BESsyness) *A structure graph $\mathcal{G} = \langle S, s_0, \rightarrow, d, r\rangle$ is* BESsy *if and only if it satisfies the following constraints:*

- *Nodes $s \in S$ such that $d(s) \in \{\top, \bot\}$ have no successor with respect to $\rightarrow$.*
- *For all nodes $s \in S$ it holds that $d(s) \in \{\wedge, \vee\}$ or $r(s)$ is defined if and only if s has a successor with respect to $\rightarrow$.*
- *For all nodes $s \in S$ that have multiple successors with respect to $\rightarrow$, it holds that $d(s) \in \{\wedge, \vee\}$.*
- *All cycles with respect to $\rightarrow$ contain at least one ranked node.*

Note that all structure graphs that are constructed from a BES using the rules in Definition 2.5 are BESsy. For all BESsy structure graphs $\mathcal{G}$ it holds that a boolean equation system can be constructed from $\mathcal{G}$.

Definition 2.7 (BES construction) *Let $\mathcal{G} = \langle S, s_0, \rightarrow, d, r\rangle$ be a BESsy structure graph. Then a BES $\mathcal{E}$ corresponding to this structure graph can be constructed as follows.*

To each ranked node $s \in S$, we associate an equation of the following form:

$$\sigma X_s = rhs_{sg}(s)$$

Here, σ is μ if $r(s)$ is odd, and ν otherwise. Variable X_s is newly introduced. Function rhs_{sg} is defined as follows:

$$rhs_{sg}(s) = \begin{cases} \bigwedge \{\phi_{sg}(s') \mid s \rightarrow s'\} & \text{if } d(u) = \wedge \\ \bigvee \{\phi_{sg}(s') \mid s \rightarrow s'\} & \text{if } d(u) = \vee \\ \phi_{sg}(s') & \text{otherwise, with } s \rightarrow s' \end{cases}$$

The function ϕ_{sg} is defined as follows:

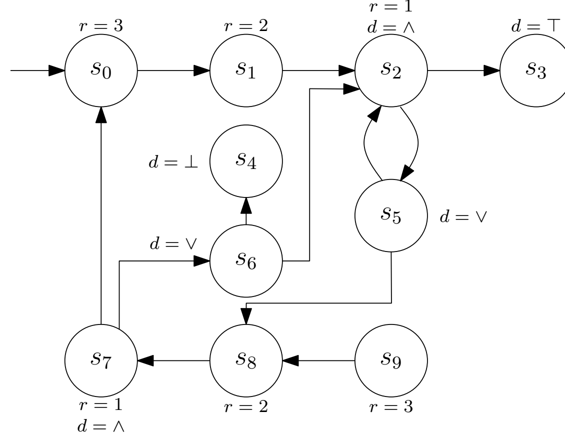
$$\phi_{sg}(s) = \begin{cases} \bigwedge \{\phi_{sg}(s') \mid s \rightarrow s'\} & \text{if } d(u) = \wedge \text{ and } s \notin dom(r) \\ \bigvee \{\phi_{sg}(s') \mid s \rightarrow s'\} & \text{if } d(u) = \vee \text{ and } s \notin dom(r) \\ true & \text{if } d(s) = \top \\ false & \text{if } d(s) = \bot \\ X_s & \text{otherwise} \end{cases}$$

The constructed set of equations must be ordered such that for all pairs of equations $(\sigma\mathcal{X}_1 = f_1), (\sigma\mathcal{X}_2 = f_2)$ such that $r(\mathcal{X}_1) > r(\mathcal{X}_2)$, it holds that $(\sigma\mathcal{X}_1 = f_1)$ occurs in the ordering before $(\sigma\mathcal{X}_2 = f_2)$.

We diverge slightly from the definition from [22], which states that nodes s with a single edge to a node s' and a decoration $d(s) = \wedge$ have $rhs_{sg}(s) = \phi_{sg}(s') \wedge \phi_{sg}(s')$ and if $s \notin dom(r)$, $\phi_{sg}(s) = \phi_{sg}(s') \wedge \phi_{sg}(s')$. The same is stated for the case in which $d(s) = \vee$. Because of the semantic equivalence of $\phi_{sg}(s')$ to $\phi_{sg}(s') \wedge \phi_{sg}(s')$ however, and the semantic nature of the used properties of structure graphs, we use the translation as given in Definition 2.7.

In [22], it is proved that a BES has an equivalent solution to the BES generated from its structure graph.

Example 5 (BES construction from structure graph) *Assume the structure graph from Example 4. We relabel the nodes for clarity:*



It can be observed that this structure graph is BESsy.

In order to transform this structure graph to a BES, we introduce one equation per ranked node. For example, the node s_2 yields the following equation:

$$\begin{aligned}\mu\mathcal{X}_{s_2} =& rhs_{sg}(s_2)\\ =&\phi_{sg}(s_3) \wedge \phi_{sg}(s_5)\\ =&true \wedge (\phi_{sg}(s_2) \vee \phi_{sg}(s_8))\\ =&true \wedge (\mathcal{X}_{s_2} \vee \mathcal{X}_{s_8})\end{aligned}$$

Repeating this process for all ranked nodes results in the following BES after ordering the equations by rank:

$$\begin{aligned}\mu\mathcal{X}_{s_0} =& \mathcal{X}_{s_1}\\ \mu\mathcal{X}_{s_9} =& \mathcal{X}_{s_8}\\ \nu\mathcal{X}_{s_1} =& \mathcal{X}_{s_2}\\ \nu\mathcal{X}_{s_8} =& \mathcal{X}_{s_7}\\ \mu\mathcal{X}_{s_2} =& true \wedge (\mathcal{X}_{s_2} \vee \mathcal{X}_{s_8})\\ \mu\mathcal{X}_{s_7} =& \mathcal{X}_{s_0} \wedge (false \vee \mathcal{X}_{s_2})\end{aligned}$$

This BES is equivalent (up to naming) to the original BES, given in Example 1.

Although it was not demonstrated in this example, it is possible to preserve the names of the recursion variables on ranked nodes through this process by choosing the names of the introduced recursion variables appropriately.

Since all BESs $\mathcal{E}$ can be transformed into structure graphs, and all structure graphs constructed this way can be transformed back into BESs that are solution equivalent to $\mathcal{E}$ (modulo renaming of recursion variables), no expressive power is lost by evaluating structure graphs instead of equation systems. This property is used in the remainder of this document.

2.2 Parametrized Boolean Equation Systems

Section 2.1 introduced boolean equation systems as a representation of a model checking problem. Parametrized boolean equation systems (PBESs), introduced in [16], are extensions of BESs with data. They are used in various model-checking problems in which data plays a role.

Definition 2.8 (PBES) *Parametrized boolean equation systems are systems of predicate fixed-point equations of the following form:*

$$\begin{aligned} PBES &::= \epsilon \\ &\mid (\sigma\mathcal{X}(d_1 : D_1, d_2 : D_2, ..., d_n : D_n) = \phi)\ PBES \end{aligned}$$

Here, σ is μ or ν, sets D_1 to D_n are data domains and ϕ is a predicate formula *(see Definition 2.9).*

In the remainder of this document, we often abstract from the number of data variables, by writing the data parameters as a single vector (so $(d_1 : D_1, d_2 : D_2, ..., d_n : D_n)$ is written as $(\vec{d} : D)$). Indexing is done using subscript (d_1) and if $\vec{d} \in D$, the type of d_i is called D_i.

Definition 2.9 (PBES predicate formula) *Predicate formulas are defined to be expressions built from the following grammar:*

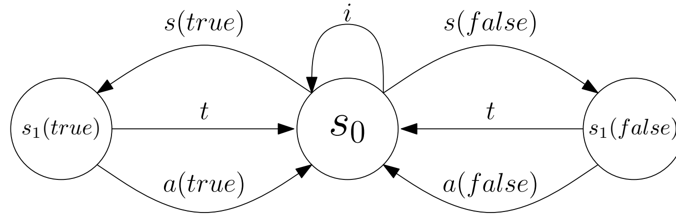
$$\phi ::= b \mid \mathcal{X}(e) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \forall \vec{v} : D.\phi \mid \exists \vec{v} : D.\phi$$

Here, b is a simple boolean expression, $\mathcal{X}$ is a predicate variable, $\vec{v}$ of type D is a vector of data variables and e is a data term.

A simple boolean expression is considered to be an expression resulting in a boolean value and containing no predicate variables. Other elements, such as quantifiers, are allowed. Constants $true$ and $false$ are considered special cases of this simple boolean expression b. We may write $b \Rightarrow \phi$ instead of $\neg b \vee \phi$ for clarity. Note that this is only allowed if the left-hand side is a simple boolean expression, because we allow no negated occurrences of recursion variables.

We also extend the definitions of *occ* and *bnd* from Definition 2.1 to PBESs. In these definitions, we ignore data. Thus $occ(\nu X(n : \mathbb{N}) = Y(n)) = \{Y\}$.

Example 6 (PBES) *We extend the description of process $\mathbb{P}$ from Example 1 with data. We assume in every send and acknowledge action, a single bit (represented by a boolean) is transferred. An acknowledge action acknowledges receiving a message by sending a copy of the bit. The following labeled transition system describes the extended system:*



We evaluate the property that after any number of steps in the system, after a bit b is sent, b must be acknowledged after any path with a finite number of i and t steps. Intuitively, this property is false, because it is possible for the system to reach s_0 in one step, without acknowledging b. From s_0 the system can enter an infinite loop of actions $s(\neg b)$ and $a(\neg b)$, so it is possible to never perform $a(b)$.

The property can be described by the following formula in modal μ calculus (extended with data, [15]):

$$\nu X.[true]X \wedge (\forall b : \mathbb{B}.[s(b)](\mu Y.(\nu Z.([i \vee t]Y \wedge [\overline{i \vee t}]Z) \vee [\overline{a(b)}]false)))$$

By combining a symbolic representation of the model with the formula, the following PBES can be constructed:

$$\begin{aligned}
\nu X_0 =& X_0 \wedge (\forall b : \mathbb{B}.X_1(b)) \wedge (\forall b : \mathbb{B}.Y_1(b)) \\
\nu X_1(b : \mathbb{B}) =& X_1(b) \\
\mu Y_0 =& Z_0 \\
\mu Y_1(b : \mathbb{B}) =& Z_1(b) \\
\nu Z_0 =& Y_0 \wedge (\forall b : \mathbb{B}.Z_1(b)) \\
\nu Z_1(b : \mathbb{B}) =& (Y_0 \wedge Z_0) \vee (\forall b' : \mathbb{B}.b \neq b' \Rightarrow false)
\end{aligned}$$

The state of the system (s_0 or s_1), which is usually transformed into a parameter, is encoded into the variable names for clarity. The parameter to state s_1 remains a parameter for the PBES. Variable X_0 corresponds to the fixed-point of X in state s_0 of the system. The other variables are named similarly.

In a similar way as with BESs, we solve a PBES in order to find the solution to the represented problem.

Definition 2.10 (PBES semantics) *In order to describe the semantics of PBESs, we first give the semantics of predicate formulas. Assume a predicate formula ϕ. Let ε be a data environment assigning a value to each data variable occurring in ϕ and let $\eta : \mathbf{X} \to D \to \mathbb{B}$ be a predicate environment, where $\mathbf{X}$ is the set of recursion variables and D the parameter space. We define the interpretation of ϕ in these environments, denoted as $[\![\phi]\!]\eta\varepsilon$, inductively as follows:*

$$\begin{aligned}
[\![b]\!]\eta\varepsilon =& [\![b]\!]\varepsilon \\
[\![\mathcal{X}(e)]\!]\eta\varepsilon =& \eta(\mathcal{X})([\![e]\!]\varepsilon) \\
[\![\phi_1 \wedge \phi_2]\!]\eta\varepsilon =& [\![\phi_1]\!]\eta\varepsilon \wedge [\![\phi_2]\!]\eta\varepsilon \\
[\![\phi_1 \vee \phi_2]\!]\eta\varepsilon =& [\![\phi_1]\!]\eta\varepsilon \vee [\![\phi_2]\!]\eta\varepsilon \\
[\![\forall \vec{v} : D'.\phi_1]\!]\eta\varepsilon =& \forall \vec{w} : D'.[\![\phi_1]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
[\![\exists \vec{v} : D'.\phi_1]\!]\eta\varepsilon =& \exists \vec{w} : D'.[\![\phi_1]\!]\eta(\varepsilon[\vec{v} := \vec{w}])
\end{aligned}$$

Here, $[\![b]\!]\varepsilon$ represents the value of the boolean expression b under assumption of the data environment ε. Notation $\varepsilon[d := v]$ replaces the value of d in environment ε with v.

The solution of a PBES $\mathcal{E}$ in the context of a predicate environment η and data environment ε is inductively defined in [16] and [29] as follows:

$$\begin{aligned}
[\![\epsilon]\!]\eta\varepsilon =& \eta \\
[\![(\sigma\mathcal{X}(\vec{d} : D) = \phi)\mathcal{E}]\!]\eta\varepsilon =& [\![\mathcal{E}]\!](\eta[\mathcal{X} := \sigma\mathcal{X}(\vec{d} : D).\phi([\![\mathcal{E}]\!]\eta\varepsilon)])
\end{aligned}$$

Here, $\sigma\mathcal{X}(\vec{d} : D).\phi([\![\mathcal{E}]\!]\eta\varepsilon)$ is a fixed-point expression, defined as follows:

$$\begin{aligned}
\mu\mathcal{X}(\vec{d} : D).\phi([\![\mathcal{E}]\!]\eta\varepsilon) =& \bigwedge \left\{ f : D \to \mathbb{B} \mid \lambda \vec{v} : D.[\![\phi]\!]([\![\mathcal{E}]\!](\eta[\mathcal{X} := f])\varepsilon)(\varepsilon[\vec{d} := \vec{v}]) \sqsubseteq f \right\} \\
\nu\mathcal{X}(\vec{d} : D).\phi([\![\mathcal{E}]\!]\eta\varepsilon) =& \bigvee \left\{ f : D \to \mathbb{B} \mid f \sqsubseteq \lambda \vec{v} : D.[\![f]\!]([\![\mathcal{E}]\!](\eta[\mathcal{X} := f])\varepsilon)(\varepsilon[\vec{d} := \vec{v}]) \right\}
\end{aligned}$$

Here, $f \sqsubseteq g$ (with $f, g : D \to \mathbb{B}$) is defined as $(\forall \vec{v} : D.f(\vec{v}) \Rightarrow g(\vec{v}))$.

In practice, these semantics are not used directly to solve PBESs, because they lead to a rather inefficient algorithm. Multiple algorithms exist for solving PBESs, but the method that is widely used consists of the process of instantiation, which transforms the system into a BES. After that, the BES can be solved using algorithms designed for BESs.

When solving a PBES, we are generally interested in the solution of a certain predicate variable instance of the system (combination of recursion variable and parameters), or in the solution for all reachable instances. We call such an instance the initial state of the system.

2.2.1 Explicit Instantiation

Explicit instantiation of a PBES to a BES, which is described in detail in [29], is done by replacing all parametrized references $\mathcal{X}(\vec{d})$ (for some parameter vector $\vec{d}$) by new, parameterless recursion variables $\mathcal{X}_{\vec{d}}$ and creating new defining equations for these variables. We only perform instantiation with reachable data elements, rather than the complete parameter space. This process is repeated until all parametrized references are instantiated. Then the resulting formulas are reordered so that the defining recursion variables for a BES equation are in the same order as their PBES counterparts. The order of the introduced equations per instantiated PBES variable can be chosen arbitrarily.

Example 7 (Explicit PBES instantiation) *Assume the PBES from Example 6:*

$$\begin{aligned}
\nu X_0 =& X_0 \wedge (\forall b : \mathbb{B}.X_1(b)) \wedge (\forall b : \mathbb{B}.Y_1(b))\\
\nu X_1(b : \mathbb{B}) =& X_1(b)\\
\mu Y_0 =& Z_0\\
\mu Y_1(b : \mathbb{B}) =& Z_1(b)\\
\nu Z_0 =& Y_0 \wedge (\forall b : \mathbb{B}.Z_1(b))\\
\nu Z_1(b : \mathbb{B}) =& (Y_0 \wedge Z_0) \vee (\forall b' : \mathbb{B}.b \neq b' \Rightarrow false)
\end{aligned}$$

We can instantiate the equation for X_0 by using a function $Inst$, performing the described actions recursively on the structure of the equation:

$$\begin{aligned}
& Inst(\nu X_0 = X_0 \wedge (\forall b : \mathbb{B}.X_1(b)) \wedge (\forall b : \mathbb{B}.Y_1(b)))\\
= \quad & \nu X_0 = Inst(X_0 \wedge (\forall b : \mathbb{B}.X_1(b)) \wedge (\forall b : \mathbb{B}.Y_1(b)))\\
= \quad & \nu X_0 = Inst(X_0) \wedge Inst(\forall b : \mathbb{B}.X_1(b)) \wedge Inst(\forall b : \mathbb{B}.Y_1(b)))\\
= \quad & \nu X_0 = X_0 \wedge Inst(X_1(false)) \wedge Inst(X_1(true)) \wedge Inst(Y_1(false)) \wedge Inst(Y_1(true))\\
= \quad & \nu X_0 = X_0 \wedge X_{1,false} \wedge X_{1,true} \wedge Y_{1,false} \wedge Y_{1,true}
\end{aligned}$$

Repeating this process on all occurrences of instantiated recursion variables yields the following BES:

$$\begin{aligned}
\nu X_0 &= X_0 \wedge X_{1,false} \wedge X_{1,true} \wedge Y_{1,false} \wedge Y_{1,true}\\
\nu X_{1,false} &= X_0\\
\nu X_{1,true} &= X_0\\
\mu Y_0 &= Z_0\\
\mu Y_{1,false} &= Z_{1,false}\\
\mu Y_{1,true} &= Z_{1,true}\\
\nu Z_0 &= Y_0 \wedge Z_{1,false} \wedge Z_{1,true}\\
\nu Z_{1,false} &= Y_0 \wedge Z_0\\
\nu Z_{1,true} &= Y_0 \wedge Z_0
\end{aligned}$$

After solving this BES (for instance by applying Gauß elimination), we observe that indeed $X_0 = false$.

3 Symbolic Instantiation

3.1 Motivation

Explicit instantiation of the complete reachable parameter space of a PBES may cause a lot of overhead in terms of both memory and processing. The reason for this is the fact that per PBES equation, many very similar BES equations may be generated. As a simple example, we can evaluate the instantiation of the equation ($\nu X(d : \mathbb{N}) = \forall n : \mathbb{N}.n < 1000 \Rightarrow X(n)$) with initial state $X(0)$. This will produce a BES with 1000 equations, each of which is a conjunction of all 1000 generated recursion variables.

We try to improve the efficiency of the instantiation process and possibly the subsequent solving of the generated BESs by making use of list decision diagrams in the instantiation. List decision diagrams (LDDs), introduced in [2] are data structures which can be used for storing and manipulating sets of similar strings. An instantiation of a PBES recursion variable may be represented by a string, treating each parameter value as a single symbol. An instantiation $\mathcal{X}(1)$ may be represented by the string $['\mathcal{X}','1']$. This way, the set of instantiated variables can be represented as a set of similar strings, which can be stored relatively efficiently in an LDD. The structure of an LDD and the operations that can be used to manipulate it are described in detail in Section 4.6. In Section 3.2, we introduce an approach for storing a complete instantiated BES using an LDD. In further sections, this approach is elaborated.

By storing the resulting BES in an LDD, a lot of memory may be saved for systems with many similar equations. Furthermore, LDDs support various set operations. By using these, it may be possible to improve the running time of BES solving algorithms by evaluating sets of equations instead of each equation separately.

3.2 Approach

Storing an arbitrary BES in an LDD efficiently is not directly possible without losing structure. The right-hand sides of a BES are formulas which may contain conjunctions and disjunctions nested arbitrarily. This nested structure cannot easily and efficiently be mapped to a set of strings. A graph, however, fits the required structure to make use of LDDs. Each node in the graph can be represented by a string, and the edge relation can be represented by a set of pairs of strings. Thus, in order to store BESs in LDDs, we can make use of a graph-based representation of these BESs.

In [20], an approach for symbolic instantiation is given. There, the chosen graph-based BES representation is a parity game ([14]). Parity games are graph-based games which have a one-to-one correspondence with BESs in simple recursive form (SRF, a normal form for BESs in which the right-hand sides of equations are fully conjunctive or fully disjunctive). Various algorithms exist to solve parity games.

In order to instantiate a PBES to a parity game, the approach starts by rewriting the PBES into parametrized parity game (PPG) form. PBESs in this normal form instantiate directly into BESs in SRF. This PPG is then interpreted as a symbolic description of a state space, where the state space itself corresponds to a parity game. Therefore, exploring the state space described by a PPG yields a parity game. The states correspond to an instantiated variable and the outgoing edges correspond to the variables that occur in the right-hand side of the equation describing the variable. Shapes (owners) and priorities of the states are determined before instantiation, and are equal for all instantiated nodes corresponding to the same PBES variable. The exploration of the state space formed by a PPG corresponds to the instantiation of the PPG.

This technique has been shown effective, vastly increasing performance of the instantiation of various systems ([20]). There are, however, disadvantages to this existing technique:

- The transformation of a PBES into PPG, as described in [20], requires a transformation of all formulas into a normal form called Bounded Quantifier Normal Form (BQNF). This normal

form consists of a sequence of bounded quantifiers, followed by a formula in a Conjunctive Normal Form (CNF) with bounded quantifiers. While all PBESs can be rewritten into BQNF ([20]), the inner CNF may cause an exponential blow-up of the formula. Consider, for instance, the two following PBES equations:

$$\nu X(\vec{d}: D) = (X_1(\vec{d}) \wedge Y_1(\vec{d})) \vee (X_2(\vec{d}) \wedge Y_2(\vec{d})) \vee ... \vee (X_n(\vec{d}) \wedge Y_n(\vec{d})) \quad (1)$$

$$\nu X_{BQNF}(\vec{d}: D) = (X_1(\vec{d}) \vee X_2(\vec{d}) \vee ... \vee X_n(\vec{d})) \wedge (Y_1(\vec{d}) \vee X_2(\vec{d}) \vee ... \vee X_n(\vec{d})) \wedge ... \wedge (Y_1(\vec{d}) \vee Y_2(\vec{d}) \vee ... \vee Y_n(\vec{d})) \quad (2)$$

These equations are equivalent, but the latter one is rewritten to BQNF. It can be seen that (2) contains disjuncts of all possible combinations of X_i and Y_i such that for all i with $1 \leqslant i \leqslant n$, either X_i or Y_i is present. As a result, there are 2^n such disjuncts of length n, while (1) contains only n conjuncts of length 2. In [26], this problem is stated in the context of the Predicate Formula Normal Form, of which BQNF is a generalization.

In addition to this explosion, another problem exists when translating the BQNF to PPG. Assume the following formula, which can be instantiated in a finite amount of time:

$$\nu X(n : \mathbb{N}) = ((n = 1) \wedge X(n + 1)) \vee ((n > 1) \wedge X(n))$$

Translating this to BQNF may yield the following:

$$\nu X(n : \mathbb{N}) = (n \geqslant 1) \wedge ((n \neq 1) \Rightarrow X(n)) \wedge ((n \leqslant 1) \Rightarrow X(n + 1)) \wedge (X(n) \vee X(n + 1))$$

Observe that when instantiating X for some $n \geqslant 1$, X must be instantiated for all $n' \geqslant n$, due to the last conjunct. Thus, by transforming a PBES to BQNF using the described method, (explicit) instantiation may no longer be possible.

- A possible solution would be to define a direct translation of a model checking problem into a PPG. While this circumvents the previous issue, such a transformation has the disadvantage that some PBES manipulations, such as *parelm* and *constelm* ([18]) become less effective. Furthermore, alternative decision problems that are encoded into PBESs, such as equivalence checking of infinite systems using branching bisimulation ([9]), do not benefit from this transformation.
- Parity games correspond syntactically to a strict subset of all BESs. There are use cases in which the desired output is not a parity game, but a BES. Although there is a direct correspondence between BESs in SRF and parity games, SRF imposes an unnecessary restriction. It is not the case that a single instantiation of a PBES equation corresponds to a single BES equation, and this change in structure may make it harder to trace problems and verification results.

In this section, we describe a generalization of the approach in [20] which aims to solve these problems by not doing the transformation to PPG form, but working on a more general type of PBES.

3.3 State-space

In order to treat instantiation of a PBES as a state-space exploration problem, we need to define the state-space itself. Whereas in the PPG method, the explored state-space is a parity game, in our generalized approach, we want to allow the resulting state-space to represent any boolean equation system.

For this, we use structure graphs (see Section 2.1.1) as the BES representation. Each ranked node in this structure graph represents an instance of a PBES equation and all edges pointing

towards these ranked nodes represent dependencies. The unranked nodes represent subformulas. The edges pointing to unranked nodes determine of which formulas they are subformulas. A single instantiation of a PBES formula may consist of multiple nodes in the explored structure graph. Unlike in the parity game approach, however, a single instantiation of a PBES equation corresponds to a single BES equation.

3.4 Clusters

In order to instantiate a PBES formula directly to a set of nodes and edges in the structure graph, we first identify subformulas of the PBES. We choose these subformulas in such a way that each instantiated subformula corresponds to a single node and its outgoing edges in the structure graph. We call these subformulas *clusters*. This means that clusters have the requirement that when instantiated they must be either $true$ or $false$, or conjunctions or disjunctions of subformulas (represented by other clusters) and recursion variables. We want a cluster to define a subformula that is as large as possible while conforming to this requirement. This corresponds to the flattening seen in Definition 2.5.

Clusters represent subformulas. Therefore, they may refer to each other, but not in a cyclic way.

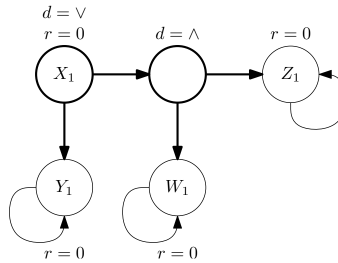
Example 8 (Cluster) *Assume the following PBES:*

$$\begin{aligned}\nu X(n:\mathbb{N}) =& Y(n) \vee (Z(n) \wedge W(n))\\ \nu Y(n:\mathbb{N}) =& Y(n)\\ \nu Z(n:\mathbb{N}) =& Z(n)\\ \nu W(n:\mathbb{N}) =& W(n)\end{aligned}$$

Observe that instantiation of X with $n = 1$ yields the following BES:

$$\begin{aligned}\nu X_1 =& Y_1 \vee (Z_1 \wedge W_1)\\ \nu Y_1 =& Y_1\\ \nu Z_1 =& Z_1\\ \nu W_1 =& W_1\end{aligned}$$

This is described by a structure graph similar to the following:

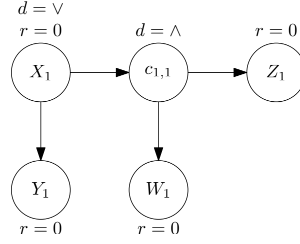


The bold elements of the image describe the part of the graph corresponding to X_1.

We can identify clusters in X as follows:

$$\begin{aligned}\nu X(n:\mathbb{N}) =& Y(n) \vee c_1(n)\\ c_1(n) =& Z(n) \wedge W(n)\end{aligned}$$

The structure graph is still the same, but now each instantiated equation corresponds to a single node:



Definition 3.1 (Cluster) *We define a cluster as a predicate (sub-)formula that instantiates to a conjunction or disjunction of a (possibly empty) set of recursion variables. We characterize a cluster as follows:*

$$\begin{aligned} CLUSTER &::= c^{\wedge}(\vec{d}:D) = CLUSTER_{\wedge} \\ &\mid c^{\vee}(\vec{d}:D) = CLUSTER_{\vee} \end{aligned}$$

$$\begin{aligned} CLUSTER_{\wedge} &::= \mathcal{X}(e) \mid c^{\vee}(e) \\ &\mid b \Rightarrow CLUSTER_{\wedge} \\ &\mid b \Rightarrow (b' \wedge CLUSTER_{\wedge}) \\ &\mid \forall \vec{v}:D'.CLUSTER_{\wedge} \\ &\mid CLUSTER_{\wedge} \wedge CLUSTER_{\wedge} \end{aligned}$$

$$\begin{aligned} CLUSTER_{\vee} &::= \mathcal{X}(e) \mid c^{\wedge}(e) \\ &\mid b \wedge CLUSTER_{\vee} \\ &\mid b \wedge (b' \Rightarrow CLUSTER_{\vee}) \\ &\mid \exists \vec{v}:D'.CLUSTER_{\vee} \\ &\mid CLUSTER_{\vee} \vee CLUSTER_{\vee} \end{aligned}$$

Here, $c^{\wedge}$ is the name of some conjunctive cluster and $c^{\vee}$ is the name of some disjunctive cluster, $\vec{d}$ is a list of data parameters of type D, b and b' are simple boolean expressions and e is a data expression.

Interesting to note is that a conjunctive cluster (following the $CLUSTER_{\wedge}$ grammar) never contains a reference to another conjunctive cluster, and that all clusters contain at least one reference to a cluster or recursion variable.

Definition 3.2 (Clustered PBES) *We characterize a clustered equation as follows:*

$$\begin{aligned} CLUSTERED_EQ &::= \sigma X(\vec{d}:D) = CRHS \\ CRHS &::= CLUSTER_{\wedge} \mid CLUSTER_{\vee} \\ &\mid b \mid b \wedge CRHS \mid b \Rightarrow CRHS \end{aligned}$$

Here, σ is a fixed-point symbol (μ or ν), X is a recursion variable, $\vec{d}$ is a parameter vector and b is a simple boolean expression (containing no recursion variables).

A clustered PBES is a sequence of parametrized boolean fixed-point equations of the form $CLUSTERED_EQ$ combined with a set of clusters of the form $CLUSTER$. Clusters may be cyclicly dependent, as long as each cycle contains a fixed-point equation.

Note that this last property is similar to the fourth constraint in the definition of BESsyness (Definition 2.6).

There may be multiple clustered PBESs corresponding to an arbitrary PBES. We try to identify clusters in a way such that the number of identified clusters is minimal. A naive approach to this may pose the problem of infinitely large instantiations, as shown in Example 9.

Example 9 (Infinite instantiations due to clustering) *Assume the following PBES equation:*

$$\nu X(n:\mathbb{N}, m:\mathbb{N}) = \forall i:\mathbb{N}.\exists j:\mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i,j))$$

Observe that the instantiation of X (with arbitrary parameters) will yield a finite BES.

A naive approach to identifying clusters might result in the following clustered PBES:

$$\begin{aligned} \nu X(n:\mathbb{N}, m:\mathbb{N}) =& \forall i:\mathbb{N}.c_0(n,m,i) \\ c_0(n:\mathbb{N}, m:\mathbb{N}, i:\mathbb{N}) =& \exists j:\mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i,j)) \end{aligned}$$

For such a cluster, instantiation of equation X (with arbitrary parameters) does not terminate, because the result is a single equation that is a conjunction of $c_0(n,m,i)$ for all $i \in \mathbb{N}$.

This problem, however, does not occur when the equations are slightly modified:

$$\begin{aligned} \nu X(n:\mathbb{N}, m:\mathbb{N}) =& \forall i:\mathbb{N}.(i \leqslant 10) \Rightarrow c_0(n,m,i) \\ c_0(n:\mathbb{N}, m:\mathbb{N}, i:\mathbb{N}) =& \exists j:\mathbb{N}.(j < 10) \wedge X(i,j) \end{aligned}$$

It can be seen that the first cluster is now always a finite conjunction, because the allowed values for i are bounded by 0 and 10. This raises the question whether it is in general possible to rewrite a PBES with a finite instantiation such that instantiation of its clustered counterpart remains finite.

A solution to this class of problems is given in Sections 3.5 and 3.6.

3.5 Guarded Normal Form

In order to make the transition from arbitrary PBESs to clustered PBESs, we introduce a normal form for PBESs named Guarded Normal Form (GNF). This form looks similar to the clustered form from Section 3.4 and acts as a characterization which helps in the identification of these clusters in PBES equations.

Definition 3.3 (GNF) *The Guarded Normal Form is a normal form for PBESs. An equation in GNF is of the following form:*

$$\begin{aligned} GNF ::=& \sigma\mathcal{X}(\vec{d}:D) = RHS \\ RHS ::=& RHS_\wedge \mid RHS_\vee \mid b \\ RHS_\wedge ::=& \mathcal{X}(e) \\ & \mid b \Rightarrow RHS_\vee \\ & \mid \forall \vec{v}:D'.b \Rightarrow RHS_\vee \\ & \mid RHS_\wedge \wedge RHS_\wedge \\ RHS_\vee ::=& \mathcal{X}(e) \\ & \mid b \wedge RHS_\wedge \\ & \mid \exists \vec{v}:D'.b \wedge RHS_\wedge \\ & \mid RHS_\vee \vee RHS_\vee \end{aligned}$$

Here, σ is one of μ (least fixed point) and ν (greatest fixed point), b is a simple boolean expression, $\mathcal{X}$ is a recursion variable and e is a data term. Recall that a simple boolean expression is an arbitrary boolean expression containing no recursion variables.

The simple boolean expressions which occur in GNF predicate formulas are used as guards for subexpressions, meaning that in order to find a solution to the system, it is only necessary to evaluate the guarded subexpression if its guards evaluate to true. Having these guards simplifies the

identification of certain dependencies within an equation, as done in Section 4.4.1. Furthermore, the guards are positioned in such a way that they help in identifying clusters in the formula.

Note that GNF puts only mild restrictions on PBESs. The main restriction is that guards must be present at symbol switches, and all subformulas that are guarded by simple boolean expressions contain at least one recursion variable.

3.5.1 Strongly Guarded Form

Solving the problem of unnecessary infinite instantiations, demonstrated in Example 9, is done by enforcing an additional property on PBESs in GNF. PBESs in GNF having this property can easily be clustered such that infinite instantiations occur only if they occur in the instantiation of the original PBES. In order to achieve this, we require the guards in the GNF, which determine whether a subexpression needs to be evaluated, to be as strong as possible.

In order to define this property, we first define helper functions $guard_\wedge$ and $guard_\vee$.

Definition 3.4 ($guard_\wedge$ and $guard_\vee$) *Functions $guard_\vee$ and $guard_\wedge$ are transformations of PBES predicate formulas, defined as follows:*

$$
\begin{aligned}
guard_\wedge(b) &= b & guard_\vee(b) &= b\\
guard_\wedge(\mathcal{X}(e)) &= true & guard_\vee(\mathcal{X}(e)) &= false\\
guard_\wedge(\phi \wedge \psi) &= guard_\wedge(\phi) \wedge guard_\wedge(\psi) & guard_\vee(\phi \wedge \psi) &= guard_\vee(\phi) \wedge guard_\vee(\psi)\\
guard_\wedge(\phi \vee \psi) &= guard_\wedge(\phi) \vee guard_\wedge(\psi) & guard_\vee(\phi \vee \psi) &= guard_\vee(\phi) \vee guard_\vee(\psi)\\
guard_\wedge(\forall \vec{v} : D.\phi) &= \forall \vec{v} : D.guard_\wedge(\phi) & guard_\vee(\forall \vec{v} : D.\phi) &= \forall \vec{v} : D.guard_\vee(\phi)\\
guard_\wedge(\exists \vec{v} : D.\phi) &= \exists \vec{v} : D.guard_\wedge(\phi) & guard_\vee(\exists \vec{v} : D.\phi) &= \exists \vec{v} : D.guard_\vee(\phi)
\end{aligned}
$$

These functions describe the transformation which replaces all occurrences of recursion variables with $true$ or $false$ respectively.

Function $guard_\wedge$ and $guard_\vee$ are defined in such a way that $guard_\wedge(\phi)$ is the strongest expression ϕ' such that for all predicate environments η, η' and data environments ε, it holds that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![\phi']\!]\eta'\varepsilon$. Similarly, $guard_\vee(\phi)$ is defined in such a way that it the weakest expression ϕ' such that for all predicate environments η, η' and data environments ε, it holds that $[\![\phi']\!]\eta\varepsilon \Rightarrow [\![\phi]\!]\eta'\varepsilon$. Lemmas 3.5 and 3.6 show that these properties indeed hold.

Lemma 3.5 *Assume a predicate formula ϕ, predicate environments η and η' and data environment ε.*

It holds that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(\phi)]\!]\eta'\varepsilon$ and $[\![guard_\vee(\phi)]\!]\eta\varepsilon \Rightarrow [\![\phi]\!]\eta'\varepsilon$.

Proof. We only prove that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(\phi)]\!]\eta'\varepsilon$. The second statement can be proved by following the same approach.

By monotonicity of PBESs ([16], Lemma 5), it holds that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(\phi)]\!]\eta\varepsilon$. We use this to reduce the claim to $[\![guard_\wedge(\phi)]\!]\eta\varepsilon \Leftrightarrow [\![guard_\wedge(\phi)]\!]\eta'\varepsilon$ for all η, η' and ε. This is trivially the case, because in the result of $guard_\wedge$, no recursion variables are present, and thus the predicate environment is not relevant in the evaluation of the semantics (see Definition 2.10).

□

In Lemma 3.6, we show that all alternatives to $guard_\wedge$ generate formulas that are at most as strong, and all alternatives to $guard_\vee$ generate formulas that are at most as weak.

Lemma 3.6 *Assume a predicate formula ϕ.*

For any predicate formula ϕ' such that for all predicate environments η, η' and data environments ε it holds that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![\phi']\!]\eta'\varepsilon$, it also holds for all η and ε that $[\![guard_\wedge(\phi)]\!]\eta\varepsilon \Rightarrow [\![\phi']\!]\eta\varepsilon$.

Symmetrically, for any predicate formula ϕ' *such that for all predicate environments* η, η' *and data environments* ε *it holds that* $[\![\phi']\!]\eta\varepsilon \Rightarrow [\![\phi]\!]\eta'\varepsilon$*, it also holds for all* η *and* ε *that* $[\![\phi']\!]\eta\varepsilon \Rightarrow [\![guard_\vee(\phi)]\!]\eta\varepsilon$*.*

Proof. We only prove the first statement, as the second statement can be proved using the same strategy.

Assume a predicate environment $\eta^\top = (\lambda X : \mathbf{X}.(\lambda d : D.true))$.

Assume a ϕ' such that $[\![\phi]\!]\eta\varepsilon \Rightarrow [\![\phi']\!]\eta'\varepsilon$ for all η, η' and ε. Then it also holds that $[\![\phi]\!]\eta^\top\varepsilon \Rightarrow [\![\phi']\!]\eta'\varepsilon$.

We first prove that for all predicate environments η and data environments ε, it holds that $[\![guard_\wedge(\phi)]\!]\eta\varepsilon \Leftrightarrow [\![\phi]\!]\eta^\top\varepsilon$. This is a proof by structural induction on ϕ. The induction hypothesis is $[\![guard_\wedge(\psi)]\!]\eta\varepsilon \Leftrightarrow [\![\psi]\!]\eta^\top\varepsilon$ for ψ smaller than ϕ.

- Case b:

$$\begin{aligned}
&[\![guard_\wedge(b)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![b]\!]\eta\varepsilon \\
\Leftrightarrow&[\![b]\!]\varepsilon \\
\Leftrightarrow&[\![b]\!]\eta^\top\varepsilon
\end{aligned}$$

- Case $\mathcal{X}(e)$:

$$\begin{aligned}
&[\![guard_\wedge(\mathcal{X}(e))]\!]\eta\varepsilon \\
\Leftrightarrow&[\![true]\!]\eta\varepsilon \\
\Leftrightarrow&true \\
\Leftrightarrow&\eta^\top(\mathcal{X})([\![e]\!]\varepsilon) \\
\Leftrightarrow&[\![\mathcal{X}(e)]\!]\eta^\top\varepsilon
\end{aligned}$$

- Case $\psi_1 \wedge \psi_2$:

$$\begin{aligned}
&[\![guard_\wedge(\psi_1 \wedge \psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(\psi_1)]\!]\eta\varepsilon \wedge [\![guard_\wedge(\psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&\{IH\} \\
&[\![\psi_1]\!]\eta^\top\varepsilon \wedge [\![\psi_2]\!]\eta^\top\varepsilon \\
\Leftrightarrow&[\![\psi_1 \wedge \psi_2]\!]\eta^\top\varepsilon
\end{aligned}$$

- Case $\psi_1 \vee \psi_2$:

$$\begin{aligned}
&[\![guard_\wedge(\psi_1 \vee \psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(\psi_1)]\!]\eta\varepsilon \vee [\![guard_\wedge(\psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&\{IH\} \\
&[\![\psi_1]\!]\eta^\top\varepsilon \vee [\![\psi_2]\!]\eta^\top\varepsilon \\
\Leftrightarrow&[\![\psi_1 \vee \psi_2]\!]\eta^\top\varepsilon
\end{aligned}$$

- Case $\forall\vec{v} : D.\psi$:

$$\begin{aligned}
&[\![guard_\wedge(\forall\vec{v} : D.\psi)]\!]\eta\varepsilon \\
\Leftrightarrow&\forall\vec{w} : D.[\![guard_\wedge(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow&\{IH\} \\
&\forall\vec{w} : D.[\![\psi]\!]\eta^\top(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow&[\![\forall\vec{v} : D.\psi]\!]\eta^\top\varepsilon
\end{aligned}$$

- Case $\exists \vec{v} : D.\psi$:

$$
\begin{aligned}
&[\![guard_\wedge(\exists \vec{v} : D.\psi)]\!]\eta\varepsilon \\
\Leftrightarrow &\exists \vec{w} : D.[\![guard_\wedge(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow &\{IH\} \\
&\exists \vec{w} : D.[\![\psi]\!]\eta^\top(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow &[\![\exists \vec{v} : D.\psi]\!]\eta^\top\varepsilon
\end{aligned}
$$

Since $[\![\phi]\!]\eta^\top\varepsilon \Rightarrow [\![\phi']\!]\eta'\varepsilon$ and $[\![guard_\wedge(\phi)]\!]\eta\varepsilon \Rightarrow [\![\phi]\!]\eta^\top\varepsilon$, it holds that $[\![guard_\wedge(\phi)]\!]\eta^\top\varepsilon \Rightarrow [\![\phi']\!]\eta^\top\varepsilon$, proving the claim.

□

Using the functions $guard_\wedge$ and $guard_\vee$, we define the property *strongly guarded*.

Definition 3.7 (Strongly Guarded Form) *An equation $(\sigma\mathcal{X}(\vec{d} : D) = \phi)$ of form GNF is strongly guarded if and only if for all simple boolean expressions b used in a conjunction $(b \wedge \phi_\wedge)$ it holds that for all predicate environments η and data environments ε, $[\![b]\!]\eta\varepsilon \Leftrightarrow [\![b \wedge guard_\wedge(\phi_\wedge)]\!]\eta\varepsilon$ and for all such expressions b' in an implication $(b' \Rightarrow \phi_\vee)$ it holds that $[\![b']\!]\eta\varepsilon \Leftrightarrow [\![b' \wedge \neg guard_\vee(\phi_\vee)]\!]\eta\varepsilon$.*

A PBES equation is in Strongly Guarded Form (SGF), if it is in GNF and is strongly guarded. A PBES is in SGF if all of its equations are in SGF.

Section 3.6 illustrates that SGF equations do not suffer from the problem of unnecessary infinite instantiations, that was shown in Example 9.

3.5.2 Transformation

As a preprocessing step to the instantiation process of a PBES, we transform the PBES into SGF.

This transformation is possible for all PBESs. We prove this by defining a transformation function F and prove that the result of the transformation is in SGF. In this transformation function, simple boolean expressions are strengthened and rewritten to act as guards.

Definition 3.8 presents transformation function F and helper functions $F_\wedge$ and $F_\vee$. Function $F_\wedge$ effectively removes the existing simple boolean expressions, replacing them by *true*. In addition to this, it adds these formulas back as guards, using the $guard_\wedge$ function to strengthen them by using information from subexpressions. The result of $F_\wedge(\phi)$ for some data expression ϕ is always of the form $RHS_\wedge$ with additional constants *true*. These constants are removed as a second step. Functions $F_\wedge$ and $F_\vee$ work in a symmetric way. Finally, we define a function F which decides whether $F_\wedge$ or $F_\vee$ should be applied.

Definition 3.8 (F) *Transformation function F is defined as follows:*

$$
\begin{aligned}
F((\sigma\mathcal{X}(\vec{d}) = \phi)\ \mathcal{E}) &= \begin{cases} (\sigma\mathcal{X}(\vec{d}) = guard_\wedge(\phi) \wedge F_\wedge(\phi))\ F(\mathcal{E}) & \text{if } \phi \in \{b, \mathcal{X}(e), \psi \wedge \upsilon, \forall \vec{v} : D.\psi\} \\ (\sigma\mathcal{X}(\vec{d}) = guard_\vee(\phi) \vee F_\vee(\phi))\ F(\mathcal{E}) & \text{otherwise} \end{cases} \\
F(\varepsilon) &= \varepsilon
\end{aligned}
$$

Here, functions $F_\wedge$ and $F_\vee$ are defined as:

- *If $occ(\phi) = \emptyset$:*

$$F_\wedge(\phi) = true \qquad\qquad F_\vee(\phi) = false$$

- *Otherwise:*

$$F_\wedge(\mathcal{X}(e)) = \mathcal{X}(e) \qquad F_\vee(\mathcal{X}(e)) = \mathcal{X}(e)$$

$$F_\wedge(\psi_1 \wedge \psi_2) = \begin{cases} F_\wedge(\psi_1) \wedge F_\wedge(\psi_2) \\ \quad \textit{if } occ(\psi_1) \neq \emptyset \wedge occ(\psi_2) \neq \emptyset \\ F_\wedge(\psi_1) \\ \quad \textit{if } occ(\psi_1) \neq \emptyset \wedge occ(\psi_2) = \emptyset \\ F_\wedge(\psi_2) \\ \quad \textit{if } occ(\psi_1) = \emptyset \wedge occ(\psi_2) \neq \emptyset \end{cases} \qquad F_\vee(\psi_1 \vee \psi_2) = \begin{cases} F_\vee(\psi_1) \vee F_\vee(\psi_2) \\ \quad \textit{if } occ(\psi_1) \neq \emptyset \wedge occ(\psi_2) \neq \emptyset \\ F_\vee(\psi_1) \\ \quad \textit{if } occ(\psi_1) \neq \emptyset \wedge occ(\psi_2) = \emptyset \\ F_\vee(\psi_2) \\ \quad \textit{if } occ(\psi_1) = \emptyset \wedge occ(\psi_2) \neq \emptyset \end{cases}$$

$$F_\wedge(\psi_1 \vee \psi_2) = \neg guard_\vee(\psi_1 \vee \psi_2) \Rightarrow F_\vee(\psi_1 \vee \psi_2) \qquad F_\vee(\psi_1 \wedge \psi_2) = guard_\wedge(\psi_1 \wedge \psi_2) \wedge F_\wedge(\psi_1 \wedge \psi_2)$$

$$F_\wedge(\forall \vec{v} : D.\psi) = \forall \vec{v} : D.\neg guard_\vee(\psi) \Rightarrow F_\vee(\psi) \qquad F_\vee(\exists \vec{v} : D.\psi) = \exists \vec{v} : D.guard_\wedge(\psi) \wedge F_\wedge(\psi)$$

$$F_\wedge(\exists \vec{v} : D.\psi) = \neg guard_\vee(\exists \vec{v} : D.\psi) \Rightarrow F_\vee(\exists \vec{v} : D.\psi) \qquad F_\vee(\forall \vec{v} : D.\psi) = guard_\wedge(\forall \vec{v} : D.\psi) \wedge F_\wedge(\forall \vec{v} : D.\psi)$$

Note that the functions from Definition 3.8 are clearly well-defined, based on the decreasing size of the input. The non-trivial case $F_\wedge(\psi_1 \vee \psi_2)$ expands to $\neg(guard_\vee(\psi_1) \vee guard_\vee(\psi_2)) \Rightarrow (F_\vee(\psi_1) \vee F_\vee(\psi_2))$.

Simplifications may be done on the simple boolean expressions occurring in the equations. The proposed optimization at this point is the removal of subexpressions that can be evaluated independently from the environments that they are in. For example, the usage of $guard_\wedge$ or $guard_\vee$ on a term such as $(\forall \vec{v} : D.\mathcal{X}(e))$ or $(\exists \vec{v} : D.\mathcal{X}(e))$ may introduce terms such as $(\forall n : \mathbb{N}.true)$ or $(\exists n : \mathbb{N}.false)$, which are independent from all data variables and predicate variables and trivially evaluate to $true$ or $false$.

Example 10 (Transformation to SGF) *We demonstrate the transformation function F described in Definition 3.8.*

Recall the PBES formula from Example 9:

$$\nu X(n : \mathbb{N}, m : \mathbb{N}) = \forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j))$$

To transform this equation into SGF, we perform the following computation.

$$\begin{aligned}
&F(\nu X(n : \mathbb{N}, m : \mathbb{N}) = \forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j)))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = guard_\wedge(\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j)))\\
&\wedge F_\wedge(\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j)))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge true))\\
&\wedge (\forall i : \mathbb{N}.\neg guard_\vee(\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j)))\\
&\Rightarrow F_\vee(\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge X(i, j))))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge true))\\
&\wedge (\forall i : \mathbb{N}.\neg(\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge false))\\
&\Rightarrow \exists j : \mathbb{N}.guard_\wedge((i > 10) \vee ((j < 10) \wedge X(i, j))) \wedge F_\wedge((i > 10) \vee ((j < 10) \wedge X(i, j))))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge true))\\
&\wedge (\forall i : \mathbb{N}.\neg(\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge false))\\
&\Rightarrow \exists j : \mathbb{N}.((i > 10) \vee ((j < 10) \wedge true)) \wedge (\neg guard_\vee((i > 10) \vee ((j < 10) \wedge X(i, j)))\\
&\Rightarrow F_\vee((i > 10) \vee ((j < 10) \wedge X(i, j)))))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge true))\\
&\wedge (\forall i : \mathbb{N}.\neg(\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge false))\\
&\Rightarrow \exists j : \mathbb{N}.((i > 10) \vee ((j < 10) \wedge true)) \wedge (\neg((i > 10) \vee ((j < 10) \wedge false))\\
&\Rightarrow F_\vee((j < 10) \wedge X(i, j))))\\
=&\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.\exists j : \mathbb{N}.(i > 10) \vee ((j < 10) \wedge true))
\end{aligned}$$

$$
\begin{aligned}
&\quad \wedge (\forall i:\mathbb{N}.\neg(\exists j:\mathbb{N}.(i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow \exists j:\mathbb{N}.((i>10) \vee ((j<10) \wedge true)) \wedge (\neg((i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow (guard_{\wedge}((j<10) \wedge X(i,j)) \wedge F_{\wedge}((j<10) \wedge X(i,j)))))\\
&= \nu X(n:\mathbb{N}, m:\mathbb{N}) = (\forall i:\mathbb{N}.\exists j:\mathbb{N}.(i>10) \vee ((j<10) \wedge true))\\
&\quad \wedge (\forall i:\mathbb{N}.\neg(\exists j:\mathbb{N}.(i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow \exists j:\mathbb{N}.((i>10) \vee ((j<10) \wedge true)) \wedge (\neg((i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow (((j<10) \wedge true) \wedge F_{\wedge}(X(i,j)))))\\
&= \nu X(n:\mathbb{N}, m:\mathbb{N}) = (\forall i:\mathbb{N}.\exists j:\mathbb{N}.(i>10) \vee ((j<10) \wedge true))\\
&\quad \wedge (\forall i:\mathbb{N}.\neg(\exists j:\mathbb{N}.(i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow \exists j:\mathbb{N}.((i>10) \vee ((j<10) \wedge true)) \wedge (\neg((i>10) \vee ((j<10) \wedge false))\\
&\quad \Rightarrow (((j<10) \wedge true) \wedge X(i,j))))\\
&= \{\mathit{Boolean\ simplification}\}\\
&\quad \nu X(n:\mathbb{N}, m:\mathbb{N}) = (\forall i:\mathbb{N}.\exists j:\mathbb{N}.(i>10) \vee (j<10)\\
&\quad \wedge (\forall i:\mathbb{N}.\neg(\exists j:\mathbb{N}.(i>10)) \Rightarrow \exists j:\mathbb{N}.((i>10) \vee (j<10)) \wedge (\neg(i>10))\\
&\quad \Rightarrow ((j<10) \wedge X(i,j))))\\
&= \{\mathit{Simplification\ of\ constants}\}\\
&\quad \nu X(n:\mathbb{N}, m:\mathbb{N}) = (\forall i:\mathbb{N}.(i \leqslant 10) \Rightarrow \exists j:\mathbb{N}.((i>10) \vee (j<10)) \wedge (i \leqslant 10)\\
&\quad \Rightarrow ((j<10) \wedge X(i,j)))
\end{aligned}
$$

In the result, it can be seen that the guard to the universal quantifier is effectively strengthened by the expression $i \leqslant 10$. Using the same naive approach to identifying clusters as in Example 9, instantiation will no longer be infinite.

3.5.3 Correctness of the Transformation

In order to prove the given transformation correct, we prove that the transformation is sound (the transformation's result is semantically equivalent to its input, Theorem 3.11) and that the transformation yields terms that are strictly in SGF (Theorem 3.13).

In addition to the proof given below, the proof is also done using the proof assistant software Coq[2] using a formalization of the PBES framework in Coq, developed by Carst Tankink[3]. Because these proof are also performed in a proof assistant, we have more certainty of the validity of the proved lemmas and theorems. In addition to this, proving properties on PBESs using this framework results in useful feedback for further development of the PBES formalization in Coq. See Appendix A for detailed information regarding the proofs in Coq.

In order to prove that transformation F is sound, we first prove a few properties of the F and *guard* functions.

While Definition 3.8 introduces a case distinction in the conjunctive case for $F_{\wedge}$ and the disjunctive case for $F_{\vee}$, the difference is merely syntactic. Lemma 3.9 proves this.

Lemma 3.9 *Assume predicate formulas ϕ_1 and ϕ_2, a predicate environment η and a data environment ε.*

The following holds:

$$
\begin{aligned}
[\![F_{\wedge}(\phi_1) \wedge F_{\wedge}(\phi_2)]\!]\eta\varepsilon &= [\![F_{\wedge}(\phi_1 \wedge \phi_2)]\!]\eta\varepsilon\\
[\![F_{\vee}(\phi_1) \vee F_{\vee}(\phi_2)]\!]\eta\varepsilon &= [\![F_{\vee}(\phi_1 \vee \phi_2)]\!]\eta\varepsilon
\end{aligned}
$$

Proof. Because of symmetry, only the first claim is proved.

[2]Coq Proof Assistant (`http://coq.inria.fr/`), version 8.4rc1, running on a 64 bits GNU/Linux machine.
[3]PBES Specification in Coq (`http://www.cs.ru.nl/~carst/`)

We make a case distinction on whether ϕ_1 and ϕ_2 contain recursion variables, mirroring Definition 3.8.

- Case $occ(\phi_1) = \emptyset \wedge occ(\phi_2) = \emptyset$:

$$\begin{aligned}
&[\![F_\wedge(\phi_1) \wedge F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![F_\wedge(\phi_1)]\!]\eta\varepsilon \wedge [\![F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![true]\!]\eta\varepsilon \wedge [\![true]\!]\eta\varepsilon \\
\Leftrightarrow&true \\
\Leftrightarrow&[\![true]\!]\eta\varepsilon \\
\Leftrightarrow&[\![F_\wedge(\phi_1 \wedge \phi_2)]\!]\eta\varepsilon
\end{aligned}$$

- Case $occ(\phi_1) = \emptyset \wedge occ(\phi_2) \neq \emptyset$:

$$\begin{aligned}
&[\![F_\wedge(\phi_1) \wedge F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![F_\wedge(\phi_1)]\!]\eta\varepsilon \wedge [\![F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![true]\!]\eta\varepsilon \wedge [\![F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&true \wedge [\![F_\wedge(\phi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![F_\wedge(\phi_1 \wedge \phi_2)]\!]\eta\varepsilon
\end{aligned}$$

- Case $occ(\phi_1) \neq \emptyset \wedge occ(\phi_2) = \emptyset$:
  Symmetric with previous case.
- Case $occ(\phi_1) \neq \emptyset \wedge occ(\phi_2) \neq \emptyset$:
  Follows directly from Definition 3.8.

□

In order to be able to prove soundness of F, it is necessary to prove that combining the *guard* and F functions results in a function of which the results are semantically equivalent to the input.

Lemma 3.10 *Assume a predicate formula ϕ. For all predicate environments η and data environments ε, it holds that $[\![\phi]\!]\eta\varepsilon \Leftrightarrow [\![guard_\wedge(\phi) \wedge F_\wedge(\phi)]\!]\eta\varepsilon$ and symmetrically, $[\![\phi]\!]\eta\varepsilon \Leftrightarrow [\![\neg guard_\vee(\phi) \Rightarrow F_\vee(\phi)]\!]\eta\varepsilon$.*

Proof. To prove this lemma, we use structural induction on ϕ. As induction hypothesis, we assume the following:

For all predicate environments η and data environments ε it holds that for all ψ, smaller than ϕ:

$$[\![\psi]\!]\eta\varepsilon \Leftrightarrow [\![guard_\wedge(\psi) \wedge F_\wedge(\psi)]\!]\eta\varepsilon \qquad (IH1)$$
$$[\![\psi]\!]\eta\varepsilon \Leftrightarrow [\![\neg guard_\vee(\psi) \Rightarrow F_\vee(\psi)]\!]\eta\varepsilon \qquad (IH2)$$

In this proof, we extensively and implicitly use the fact that $[\![\psi_1 \wedge \psi_2]\!]\eta\varepsilon = [\![\psi_1]\!]\eta\varepsilon \wedge [\![\psi_2]\!]\eta\varepsilon$ and $[\![\psi_1 \vee \psi_2]\!]\eta\varepsilon = [\![\psi_1]\!]\eta\varepsilon \vee [\![\psi_2]\!]\eta\varepsilon$ for all environments η and ε, which is stated in Definition 2.10.

The proofs for $[\![\phi]\!]\eta\varepsilon \Leftrightarrow [\![\neg guard_\vee(\phi) \Rightarrow F_\vee(\phi)]\!]\eta\varepsilon$ are omitted, because of symmetry with the given proof.

- If $occ(\phi) = \emptyset$:

$$
\begin{aligned}
& [\![\phi]\!]\eta\varepsilon \\
\Leftrightarrow & \{occ(\phi) = \emptyset\} \\
& [\![guard_{\wedge}(\phi)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\phi)]\!]\eta\varepsilon \wedge true \\
\Leftrightarrow & \{occ(\phi) = \emptyset\} \\
& [\![guard_{\wedge}(\phi)]\!]\eta\varepsilon \wedge [\![F_{\wedge}(\phi)]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \mathcal{X}(e)$:

$$
\begin{aligned}
& [\![\mathcal{X}(e)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![true]\!]\eta\varepsilon \wedge [\![\mathcal{X}(e)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\mathcal{X}(e))]\!]\eta\varepsilon \wedge [\![F_{\wedge}(\mathcal{X}(e))]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\mathcal{X}(e)) \wedge F_{\wedge}(\mathcal{X}(e))]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \psi_1 \wedge \psi_2$:

$$
\begin{aligned}
& [\![\psi_1 \wedge \psi_2]\!]\eta\varepsilon \\
\Leftrightarrow & [\![\psi_1]\!]\eta\varepsilon \wedge [\![\psi_2]\!]\eta\varepsilon \\
\Leftrightarrow & \{IH1\} \\
& [\![(guard_{\wedge}(\psi_1) \wedge F_{\wedge}(\psi_1))]\!]\eta\varepsilon \wedge [\![(guard_{\wedge}(\psi_2) \wedge F_{\wedge}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & [\![(guard_{\wedge}(\psi_1) \wedge F_{\wedge}(\psi_1)) \wedge (guard_{\wedge}(\psi_2) \wedge F_{\wedge}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & [\![(guard_{\wedge}(\psi_1) \wedge guard_{\wedge}(\psi_2)) \wedge (F_{\wedge}(\psi_1) \wedge F_{\wedge}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & \{Lemma\ 3.9\} \\
& [\![guard_{\wedge}(\psi_1 \wedge \psi_2) \wedge F_{\wedge}(\psi_1 \wedge \psi_2)]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \psi_1 \vee \psi_2$:

$$
\begin{aligned}
& [\![\psi_1 \vee \psi_2]\!]\eta\varepsilon \\
\Leftrightarrow & \{Lemma\ 3.5\} \\
& [\![\psi_1 \vee (guard_{\wedge}(\psi_1) \wedge \psi_2) \vee (guard_{\wedge}(\psi_2) \wedge \psi_1) \vee \psi_2]\!]\eta\varepsilon \\
\Leftrightarrow & \{IH1\} \\
& [\![(guard_{\wedge}(\psi_1) \wedge F_{\wedge}(\psi_1)) \vee (guard_{\wedge}(\psi_1) \wedge \psi_2) \\
& \vee (guard_{\wedge}(\psi_2) \wedge \psi_1) \vee (guard_{\wedge}(\psi_2) \wedge F_{\wedge}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & [\![(guard_{\wedge}(\psi_1) \wedge guard_{\wedge}(\psi_1) \wedge F_{\wedge}(\psi_1)) \vee (guard_{\wedge}(\psi_1) \wedge \psi_2) \vee (guard_{\wedge}(\psi_2) \wedge \psi_1) \\
& \vee (guard_{\wedge}(\psi_2) \wedge guard_{\wedge}(\psi_2) \wedge F_{\wedge}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & \{IH1\} \\
& [\![(guard_{\wedge}(\psi_1) \wedge \psi_1) \vee (guard_{\wedge}(\psi_1) \wedge \psi_2) \vee (guard_{\wedge}(\psi_2) \wedge \psi_1) \vee (guard_{\wedge}(\psi_2) \wedge \psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![(guard_{\wedge}(\psi_1) \vee guard_{\wedge}(\psi_2)) \wedge (\psi_1 \vee \psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow & \{IH2\} \\
& [\![(guard_{\wedge}(\psi_1) \vee guard_{\wedge}(\psi_2)) \wedge (guard_{\vee}(\psi_1) \vee guard_{\vee}(\psi_2) \vee F_{\vee}(\psi_1) \vee F_{\vee}(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & \{Lemma\ 3.9\} \\
& [\![(guard_{\wedge}(\psi_1) \vee guard_{\wedge}(\psi_2)) \wedge (guard_{\vee}(\psi_1 \vee \psi_2) \vee F_{\vee}(\psi_1 \vee \psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\psi_1 \vee \psi_2) \wedge F_{\wedge}(\psi_1 \vee \psi_2)]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \forall \vec{v} : D.\psi$:

$$
\begin{aligned}
& [\![\forall \vec{v} : D.\psi]\!]\eta\varepsilon \\
\Leftrightarrow & \forall \vec{w} : D.[\![\psi]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \{IH1\} \\
& \forall \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge F_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \forall \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge guard_{\wedge}(\psi) \wedge F_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \{IH1\} \\
& \forall \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge \psi]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \{IH2\} \\
& \forall \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge (guard_{\vee}(\psi) \vee F_{\vee}(\psi))]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \forall \vec{w} : D.[\![guard_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \wedge \forall \vec{w} : D.[\![(guard_{\vee}(\psi) \vee F_{\vee}(\psi))]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & [\![guard_{\wedge}(\forall \vec{v} : D.\psi) \wedge F_{\wedge}(\forall \vec{v} : D.\psi)]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \exists \vec{v} : D.\psi$:

$$
\begin{aligned}
& [\![\exists \vec{v} : D.\psi]\!]\eta\varepsilon \\
\Leftrightarrow & \exists \vec{w} : D.[\![\psi]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \{IH1\} \\
& \exists \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge F_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \exists \vec{w} : D.[\![guard_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \wedge \exists \vec{w} : D.[\![guard_{\wedge}(\psi) \wedge F_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \exists \vec{w} : D.[\![guard_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \wedge \exists \vec{w} : D.[\![\psi]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & \{IH2\} \\
& \exists \vec{w} : D.[\![guard_{\wedge}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \wedge \exists \vec{w} : D.[\![\neg guard_{\vee}(\psi) \Rightarrow F_{\vee}(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\
\Leftrightarrow & [\![\exists \vec{v} : D.guard_{\wedge}(\psi)]\!]\eta\varepsilon \wedge [\![\exists \vec{v} : D.\neg guard_{\vee}(\psi) \Rightarrow F_{\vee}(\psi)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\exists \vec{v} : D.\psi)]\!]\eta\varepsilon \wedge [\![F_{\wedge}(\exists \vec{v} : D.\psi)]\!]\eta\varepsilon \\
\Leftrightarrow & [\![guard_{\wedge}(\exists \vec{v} : D.\psi) \wedge F_{\wedge}(\exists \vec{v} : D.\psi)]\!]\eta\varepsilon
\end{aligned}
$$

□

Theorem 3.11 *Assume a PBES $\mathcal{E}$. It holds that $[\![\mathcal{E}]\!]\eta\varepsilon = [\![F(\mathcal{E})]\!]\eta\varepsilon$ for all predicate environments η and data environments ε.*

Proof.

We first use structural induction on $\mathcal{E}$. If $\mathcal{E} = \epsilon$, the claim trivially holds.

We use the fact that if for all η and ε it holds that $[\![\phi]\!]\eta\varepsilon = [\![\phi']\!]\eta\varepsilon$, it holds for all η and ε that $[\![(\sigma\mathcal{X}(\vec{d} : D) = \phi)\mathcal{E}]\!]\eta\varepsilon = [\![(\sigma\mathcal{X}(\vec{d} : D) = \phi')\mathcal{E}]\!]\eta\varepsilon$, which follows directly from Corollary 29 of [16].

If $\mathcal{E} = (\sigma\mathcal{X}(\vec{d}) = \phi)\mathcal{E}'$, the claim holds by induction if and only if $[\![\phi]\!]\eta\varepsilon = [\![guard_{\wedge}(\phi) \wedge F_{\wedge}(\phi)]\!]\eta\varepsilon$ for conjunctive ϕ and $[\![\phi]\!]\eta\varepsilon = [\![guard_{\wedge}(\phi) \vee F_{\vee}(\phi)]\!]\eta\varepsilon$ for disjunctive ϕ. Lemma 3.10 shows that this holds.

Thus, transformation function F is sound. □

In order to prove that the results of function F are of the form SGF, we first prove a lemma, which is used for showing that the results of F are strongly guarded.

Lemma 3.12 *For all predicate formulas ϕ, predicate environments η and data environments ε, it holds that $[\![guard_{\wedge}(\phi)]\!]\eta\varepsilon \Leftrightarrow [\![guard_{\wedge}(\phi) \wedge guard_{\wedge}(F_{\wedge}(\phi))]\!]\eta\varepsilon$ and $[\![guard_{\vee}(\phi)]\!]\eta\varepsilon \Leftrightarrow [\![guard_{\vee}(\phi) \vee guard_{\vee}(F_{\vee}(\phi))]\!]\eta\varepsilon$.*

Proof. We prove only the first statement by induction on ϕ. The proof for the second statement is symmetric. We use the following induction hypothesis:

For all ψ smaller than ϕ, predicate environments η and data environments ε, it holds that $[\![guard_\wedge(\psi)]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(F_\wedge(\psi))]\!]\eta\varepsilon$. (IH1)

As a first step, we define prove that under assumption of (IH1), the following holds:
$[\![guard_\wedge(\psi)]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(guard_\vee(\psi) \vee F_\vee(\psi))]\!]\eta\varepsilon$. (H2)
Note that ψ is smaller than ϕ.

To prove (H2), we use a nested induction with the following hypothesis:
For all predicate formulas υ smaller than ψ, predicate environments η and data environments ε, it holds that $[\![guard_\wedge(\upsilon)]\!]\eta\varepsilon \Rightarrow [\![guard_\wedge(guard_\vee(\upsilon)\vee F_\vee(\upsilon))]\!]\eta\varepsilon$. (IH2)

We use a case distinction on ψ.

- Case b:

$$\begin{aligned}
&[\![guard_\wedge(b)]\!]\eta\varepsilon\\
\Rightarrow&[\![b]\!]\eta\varepsilon \vee [\![true]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(b))]\!]\eta\varepsilon \vee [\![guard_\wedge(F_\vee(b))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(b) \vee F_\vee(b))]\!]\eta\varepsilon
\end{aligned}$$

- Case $\mathcal{X}(e)$:

$$\begin{aligned}
&[\![guard_\wedge(\mathcal{X}(e))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![false]\!]\eta\varepsilon \vee [\![true]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(\mathcal{X}(e)))]\!]\eta\varepsilon \vee [\![guard_\wedge(F_\vee(true))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(\mathcal{X}(e)))]\!]\eta\varepsilon \vee [\![guard_\wedge(F_\vee(\mathcal{X}(e)))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(\mathcal{X}(e)) \vee F_\vee(\mathcal{X}(e)))]\!]\eta\varepsilon
\end{aligned}$$

- Case $\upsilon_1 \wedge \upsilon_2$:

$$\begin{aligned}
&[\![guard_\wedge(\upsilon_1 \wedge \upsilon_2)]\!]\eta\varepsilon\\
\Rightarrow&\{(IH1)\}\\
&[\![guard_\wedge(\upsilon_1 \wedge \upsilon_2) \wedge guard_\wedge(F_\wedge(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon\\
\Rightarrow&\{Lemma\ 3.5\}\\
&[\![guard_\wedge(guard_\wedge(\upsilon_1 \wedge \upsilon_2)) \wedge guard_\wedge(F_\wedge(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\wedge(\upsilon_1 \wedge \upsilon_2) \wedge F_\wedge(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(F_\vee(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon\\
\Rightarrow&[\![guard_\wedge(guard_\vee(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon \vee [\![guard_\wedge(F_\vee(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon\\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(\upsilon_1 \wedge \upsilon_2) \vee F_\vee(\upsilon_1 \wedge \upsilon_2))]\!]\eta\varepsilon
\end{aligned}$$

- Case $\upsilon_1 \vee \upsilon_2$:

$$\begin{aligned}
&[\![guard_\wedge(\upsilon_1 \vee \upsilon_2)]\!]\eta\varepsilon\\
\Rightarrow&\{(IH1)\}\\
&[\![guard_\wedge(F_\wedge(\upsilon_1 \vee \upsilon_2))]\!]\eta\varepsilon\\
\Leftrightarrow&\{Definition\ 3.8\}\\
&[\![guard_\wedge(guard_\vee(\upsilon_1 \vee \upsilon_2) \vee F_\vee(\upsilon_1 \vee \upsilon_2))]\!]\eta\varepsilon
\end{aligned}$$

- Case $\forall \vec{v} : D.\upsilon$:

$$
\begin{aligned}
&[\![guard_\wedge(\forall \vec{v} : D.\upsilon)]\!]\eta\varepsilon \\
\Leftrightarrow&\forall \vec{d} : D.[\![guard_\wedge(\upsilon)]\!]\eta(\varepsilon[\vec{v} := \vec{d}]) \\
\Leftrightarrow&\{(IH2)\} \\
&(\forall \vec{d} : D.[\![guard_\wedge(\upsilon)]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \wedge (\forall \vec{d} : D.[\![guard_\wedge(guard_\vee(\upsilon) \vee F_\vee(\upsilon))]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \\
\Leftrightarrow&(\forall \vec{d} : D.[\![guard_\wedge(\upsilon)]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \wedge (\forall \vec{d} : D.[\![guard_\wedge(F_\wedge(\upsilon))]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \\
\Rightarrow&\{Lemma\ 3.5\} \\
&(\forall \vec{d} : D.[\![guard_\wedge(guard_\wedge(\upsilon))]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \wedge (\forall \vec{d} : D.[\![guard_\wedge(F_\wedge(\upsilon))]\!]\eta(\varepsilon[\vec{v} := \vec{d}])) \\
\Leftrightarrow&[\![\forall \vec{v} : D.guard_\wedge(guard_\wedge(\upsilon))]\!]\eta\varepsilon \wedge [\![\forall \vec{v} : D.guard_\wedge(F_\wedge(\upsilon))]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(\forall \vec{v} : D.guard_\wedge(\upsilon)) \wedge guard_\wedge(\forall \vec{v} : D.F_\wedge(\upsilon))]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(F_\vee(\forall \vec{v} : D.\upsilon))]\!]\eta\varepsilon \\
\Rightarrow&[\![guard_\wedge(guard_\vee(\forall \vec{v} : D.\upsilon))]\!]\eta\varepsilon \vee [\![F_\vee(\forall \vec{v} : D.\upsilon))]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(guard_\vee(\forall \vec{v} : D.\upsilon) \vee F_\vee(\forall \vec{v} : D.\upsilon))]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\phi = \exists \vec{v} : D.\upsilon$:

$$
\begin{aligned}
&[\![guard_\wedge(\exists \vec{v} : D.\upsilon)]\!]\eta\varepsilon \\
\Rightarrow&\{(IH1)\} \\
&[\![guard_\wedge(F_\wedge(\exists \vec{v} : D.\upsilon))]\!]\eta\varepsilon \\
\Leftrightarrow&\{Definition\ 3.8\} \\
&[\![guard_\wedge(guard_\vee(\exists \vec{v} : D.\upsilon) \vee F_\vee(\exists \vec{v} : D.\upsilon))]\!]\eta\varepsilon
\end{aligned}
$$

Having proved (H2), we can use it in the remainder of the proof.

- Case b:

$$
\begin{aligned}
&[\![guard_\wedge(b)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![true]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(true)]\!]\eta\varepsilon \\
\Leftrightarrow&\{F_\wedge(b) = true\} \\
&[\![guard_\wedge(F_\wedge(b))]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\mathcal{X}(e)$:

$$
\begin{aligned}
&[\![guard_\wedge(\mathcal{X}(e))]\!]\eta\varepsilon \\
\Leftrightarrow&\{F_\wedge(\mathcal{X}(e)) = \mathcal{X}(e)\} \\
&[\![guard_\wedge(F_\wedge(\mathcal{X}(e)))]\!]\eta\varepsilon
\end{aligned}
$$

- Case $\psi_1 \wedge \psi_2$:

$$
\begin{aligned}
&[\![guard_\wedge(\psi_1 \wedge \psi_2)]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(\psi_1)]\!]\eta\varepsilon \wedge [\![guard_\wedge(\psi_2)]\!]\eta\varepsilon \\
\Rightarrow&\{(IH1)\} \\
\Leftrightarrow&[\![guard_\wedge(F_\wedge(\psi_1))]\!]\eta\varepsilon \wedge [\![guard_\wedge(F_\wedge(\psi_2))]\!]\eta\varepsilon \\
\Leftrightarrow&[\![guard_\wedge(F_\wedge(\psi_1) \wedge F_\wedge(\psi_2))]\!]\eta\varepsilon
\end{aligned}
$$

There are three cases. Either $occ(\psi_1) \neq \emptyset$ or $occ(\psi_2) \neq \emptyset$ or both. Using an argument similar to the proof for Lemma 3.9, we find that in all three cases, $[\![guard_\wedge(F_\wedge(\psi_1)) \wedge guard_\wedge(F_\wedge(\psi_2))]\!]\eta\varepsilon = [\![guard_\wedge(F_\wedge(\psi_1 \wedge \psi_2))]\!]\eta\varepsilon$. Thus:

$$\begin{aligned} & [\![guard_\wedge(F_\wedge(\psi_1) \wedge F_\wedge(\psi_2))]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(F_\wedge(\psi_1 \wedge \psi_2))]\!]\eta\varepsilon \end{aligned}$$

- Case $\psi_1 \vee \psi_2$:

$$\begin{aligned} & [\![guard_\wedge(\psi_1 \vee \psi_2)]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(\psi_1)]\!]\eta\varepsilon \vee [\![guard_\wedge(\psi_2)]\!]\eta\varepsilon \\ \Rightarrow & \{(H2)\} \\ & [\![guard_\wedge(guard_\vee(\psi_1) \vee F_\vee(\psi_1))]\!]\eta\varepsilon \vee [\![guard_\wedge(guard_\vee(\psi_2) \vee F_\vee(\psi_2))]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(guard_\vee(\psi_1 \vee \psi_2) \vee F_\vee(\psi_1 \vee \psi_2))]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(F_\wedge(\psi_1 \vee \psi_2))]\!]\eta\varepsilon \end{aligned}$$

- Case $\forall \vec{v} : D.\psi$:

$$\begin{aligned} & [\![guard_\wedge(\forall \vec{v} : D.\psi)]\!]\eta\varepsilon \\ \Leftrightarrow & \forall \vec{w} : D.[\![guard_\wedge(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\ \Rightarrow & \{(IH1)\} \\ & \forall \vec{w} : D.[\![guard_\wedge(F_\wedge(\psi))]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\ \Leftrightarrow & [\![\forall \vec{v} : D.guard_\wedge(F_\wedge(\psi))]\!]\eta\varepsilon \end{aligned}$$

- Case $\exists \vec{v} : D.\psi$:

$$\begin{aligned} & [\![guard_\wedge(\exists \vec{v} : D.\psi)]\!]\eta\varepsilon \\ \Leftrightarrow & [\![\exists \vec{v} : D.guard_\wedge(\psi)]\!]\eta\varepsilon \\ \Leftrightarrow & \exists \vec{w} : D.[\![guard_\wedge(\psi)]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\ \Rightarrow & \{(H2)\} \\ & \exists \vec{w} : D.[\![guard_\wedge(guard_\vee(\psi) \vee F_\vee(\psi))]\!]\eta(\varepsilon[\vec{v} := \vec{w}]) \\ \Leftrightarrow & [\![\exists \vec{v} : D.guard_\wedge(guard_\vee(\psi) \vee F_\vee(\psi))]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(guard_\vee(\exists \vec{v} : D.\psi) \vee F_\vee(\exists \vec{v} : D.\psi))]\!]\eta\varepsilon \\ \Leftrightarrow & [\![guard_\wedge(F_\wedge(\exists \vec{v} : D.\psi))]\!]\eta\varepsilon \end{aligned}$$

The proof for the lemma follows directly from (IH1). □

Theorem 3.13 *The result of transformation function F on any PBES is of the form SGF.*

Proof. We prove that $F(\sigma\mathcal{X}(\vec{d}) = \phi)$ for PBES equation $(\sigma\mathcal{X}(\vec{d}) = \phi)$ yields an equation $(\sigma\mathcal{X}(\vec{d}) = \phi')$ which is of the form GNF. Furthermore we prove that ϕ' is strongly guarded.

First, we observe that the results of the transformations $guard_\wedge$ and $guard_\vee$ do not include any recursion variables, and can thus be interpreted as simple boolean expressions (b).

If ϕ is a simple boolean expression, then $F(\sigma\mathcal{X}(\vec{d}) = \phi)$ is trivially of the correct form. If it is not, we make a case distinction.

If ϕ is conjunctive:

$$F(\phi) = F_\wedge(\phi) = (\sigma X(\vec{d}) = guard_\wedge(\phi) \wedge F_\wedge(\phi))$$

Because $guard_\wedge(\phi)$ is of the form b, $F(\sigma\mathcal{X}(\vec{d}) = \phi)$ is of the form SGF if and only if $F_\wedge(\phi)$ is of the form RHS and $guard_\wedge(\phi) \Leftrightarrow guard_\wedge(\phi) \wedge guard_\wedge(F_\wedge(\phi))$, which is true by Lemma 3.12.

The case in which ϕ is disjunctive is symmetric.

In order to show that $F_\wedge(\phi)$ and $F_\vee(\phi)$ are of the form RHS, it is proved that for all ϕ that are not simple boolean expressions (b, $occ(\phi) = \emptyset$), the results of $F_\wedge(\phi)$ and $F_\vee(\phi)$ are of the form $RHS_\wedge$ and $RHS_\vee$ respectively.

We only prove that for all PBESs ϕ it holds that $F_\wedge(\phi)$ is of the form $RHS_\wedge$, as the proof for $F_\vee$ is symmetric.

We use induction on the structure of ϕ. The induction hypotheses are:

$$F_\wedge(\psi) \text{ is of the form } RHS_\wedge \text{ for } \psi < \phi \text{ if } occ(\psi) \neq \emptyset \qquad (IH1)$$
$$F_\vee(\psi) \text{ is of the form } RHS_\vee \text{ for } \psi < \phi \text{ if } occ(\psi) \neq \emptyset \qquad (IH2)$$

- Case $\phi = \mathcal{X}(e)$:

$$\begin{aligned} & F_\wedge(\mathcal{X}(e)) \\ = & \mathcal{X}(e) \end{aligned}$$

  This is trivially of the form $RHS_\wedge$.

- Case $\phi = \psi_1 \wedge \psi_2$:

$$\begin{aligned} & F_\wedge(\psi_1 \wedge \psi_2) \\ = & \begin{cases} F_\wedge(\psi_1) & \text{if } occ(\psi_2) = \emptyset \\ F_\wedge(\psi_2) & \text{if } occ(\psi_1) = \emptyset \\ F_\wedge(\psi_1) \wedge F_\wedge(\psi_2) & \text{otherwise} \end{cases} \end{aligned}$$

  If $occ(\psi_1) = \emptyset$, the result is $F_\wedge(\psi_2)$. Because $occ(\psi_1 \wedge \psi_2) \neq \emptyset$, It must hold that $occ(\psi_2) \neq \emptyset$, and the induction hypothesis can be used.

  The case in which $occ(\psi_2) = \emptyset$ is symmetrical.

  If $occ(\psi_1) \neq \emptyset$ and $occ(\psi_2) \neq \emptyset$, the result is a conjunction of two terms of the form $RHS_\wedge$ (by the induction hypothesis), which is also in $RHS_\wedge$.

- Case $\phi = \psi_1 \vee \psi_2$:

$$\begin{aligned} & F_\wedge(\psi_1 \vee \psi_2) \\ = & \neg guard_\vee(\psi_1 \vee \psi_2) \Rightarrow F_\vee(\psi_1 \vee \psi_2) \end{aligned}$$

  Using a similar argument to the one in the previous case, we find that $F_\vee(\psi_1 \vee \psi_2)$ is of the form $RHS_\vee$.

  Because $guard_\vee(\psi_1 \vee \psi_2)$ is a simple boolean expression, the result is an implication between a simple boolean expression and a subformula of the form $RHS_\vee$, which is a legal conjunct in $RHS_\wedge$.

  Furthermore, by Lemma 3.12, the requirement that $guard_\vee(\psi_1 \vee \psi_2) \Leftrightarrow guard_\vee(\psi_1 \vee \psi_2) \vee guard_\vee(F_\vee(\psi_1 \vee \psi_2))$ is satisfied, so the expression is strongly guarded.

- Case $\phi = \forall \vec{v} : D.\psi$:

$$\begin{aligned} & F_\wedge(\forall \vec{v} : D.\psi) \\ = & \forall \vec{v} : D.\neg guard_\vee(\psi) \Rightarrow F_\vee(\psi) \end{aligned}$$

  Because $guard_\vee(\psi)$ is a simple boolean expression, so is $\neg guard_\vee(\psi)$.

Since $occ(\forall \vec{v} : D.\psi) \neq \emptyset$, it holds that $occ(\psi) \neq \emptyset$. By $(IH2)$, $F_\vee(\psi)$ is of the form $RHS_\vee$, and thus the whole expression is of the form $RHS_\wedge$.

Furthermore, by Lemma 3.12, the requirement that $guard_\vee(\psi) \Leftrightarrow guard_\vee(\psi) \vee guard_\vee(F_\vee(\psi))$ is satisfied, so the expression is strongly guarded.

- Case $\phi = \exists \vec{v} : D.\psi$:

$$\begin{aligned}
&F_\wedge(\exists \vec{v} : D.\psi) \\
=&\neg guard_\vee(\exists \vec{v} : D.\psi) \Rightarrow F_\vee(\exists \vec{v} : D.\psi) \\
=&(\neg \exists \vec{v} : D.guard_\vee(\psi)) \Rightarrow (\exists \vec{v} : D.guard_\wedge(\psi) \wedge F_\wedge(\psi))
\end{aligned}$$

Since $occ(\exists \vec{v} : D.\psi) \neq \emptyset$, it holds that $occ(\psi) \neq \emptyset$.

Because $guard_\wedge(\psi)$ results in a simple boolean expression and by $(IH1)$, $F_\wedge(\psi)$ is of the form $RHS_\wedge$, it holds that $(\exists \vec{v} : D.guard_\wedge(\psi) \wedge F_\wedge(\psi))$ is of the form $RHS_\vee$.

Since $guard_\vee(\psi)$ is a simple boolean expression, the expression $(\neg \exists \vec{v} : D.guard_\vee(\psi))$ is one as well. Therefore, the whole expression is of the form $RHS_\vee$.

Furthermore, Lemma 3.12 satisfies the following additional requirements:

$$\begin{aligned}
guard_\wedge(\psi) \Leftrightarrow& guard_\wedge(\psi) \wedge guard_\wedge(F_\wedge(\psi_1 \vee \psi_2)) \\
guard_\vee(\exists \vec{d} : D.guard_\vee(\psi)) \Leftrightarrow& guard_\vee(\exists \vec{d} : D.guard_\vee(\psi)) \vee guard_\vee(F_\vee(\psi))
\end{aligned}$$

As a consequence, the expression is strongly guarded.

□

3.6 Clustered GNF

After transforming a PBES into Guarded Normal Form, clusters can be identified through a simple process. In this process, we split off a subformula from an equation in the form of a cluster only if keeping it would cause the equation to not to be of the required form (see Section 3.4). Furthermore, the strong guards from SGF are retained throughout the clustering, eliminating the possibility of unnecessary infinite instantiations when clustering a PBES in SGF. This problem was shown in Example 9.

Rewriting a system in GNF into a clustered representation can be done by applying a simple transformation function to the system.

Definition 3.14 (Cluster identification function *Cluster***)** *Assume a PBES $\mathcal{E}$ in GNF. We transform $\mathcal{E}$ into a clustered PBES by calculating $Cluster(\mathcal{E})$, where Cluster is defined as follows:*

$$\begin{aligned}
Cluster(\epsilon) =&\epsilon \\
Cluster((\sigma \mathcal{X}(\vec{d} : D) = \phi)\mathcal{E}) =&(\sigma \mathcal{X}(\vec{d} : D) = Cluster(\phi, \vec{d} : D))\ Cluster(\mathcal{E})
\end{aligned}$$

$$\begin{aligned}
Cluster(b, \vec{d} : D) =&b \\
Cluster(\mathcal{X}(e), \vec{d} : D) =&\mathcal{X}(e) \\
Cluster(b \wedge \psi, \vec{d} : D) =&b \wedge Cluster(\psi, \vec{d} : D) \\
Cluster(b \Rightarrow \psi, \vec{d} : D) =&b \Rightarrow Cluster(\psi, \vec{d} : D) \\
Cluster(\psi_1 \wedge \psi_2, \vec{d} : D) =&Cluster_\wedge(\psi_1 \wedge \psi_2, \vec{d} : D) \qquad (occ(\psi_1) \neq \emptyset) \\
Cluster(\psi_1 \vee \psi_2, \vec{d} : D) =&Cluster_\vee(\psi_1 \vee \psi_2, \vec{d} : D) \qquad (occ(\psi_1) \neq \emptyset) \\
Cluster(\forall \vec{v} : D'.\psi, \vec{d} : D) =&Cluster_\wedge(\forall \vec{v} : D'.\psi, (\vec{d} +\!\!+ \vec{v} : D \times D')) \\
Cluster(\exists \vec{v} : D'.\psi, \vec{d} : D) =&Cluster_\vee(\exists \vec{v} : D'.\psi, (\vec{d} +\!\!+ \vec{v} : D \times D'))
\end{aligned}$$

$$
\begin{aligned}
Cluster_\wedge(b, \vec{d} : D) =& b \\
Cluster_\wedge(\mathcal{X}(e), \vec{d} : D) =& \mathcal{X}(e) \\
Cluster_\wedge(b \Rightarrow \psi, \vec{d} : D) =& b \Rightarrow Cluster_\wedge(\psi, \vec{d} : D) \\
Cluster_\wedge(\psi_1 \wedge \psi_2, \vec{d} : D) =& Cluster_\wedge(\psi_1, \vec{d} : D) \wedge Cluster_\wedge(\psi_2, \vec{d} : D) \\
Cluster_\wedge(\psi_1 \vee \psi_2, \vec{d} : D) =& c_*(\vec{d}) \\
& \textit{where } (c_*(\vec{d} : D) = Cluster_\vee(\psi_1 \vee \psi_2, \vec{d} : D)) \textit{ is a fresh cluster} \\
Cluster_\wedge(\forall \vec{v} : D'.\psi, \vec{d} : D) =& \forall \vec{v} : D'.Cluster_\wedge(\psi, (\vec{d} +\!\!+ \vec{v} : D \times D')) \\
Cluster_\wedge(\exists \vec{v} : D'.\psi, \vec{d} : D) =& c_*(\vec{d}) \\
& \textit{where } (c_*(\vec{d} : D) = Cluster_\vee(\exists \vec{v} : D'.\psi)) \textit{ is a fresh cluster}
\end{aligned}
$$

$$
\begin{aligned}
Cluster_\vee(b, \vec{d} : D) =& b \\
Cluster_\vee(\mathcal{X}(e), \vec{d} : D) =& \mathcal{X}(e) \\
Cluster_\vee(b \wedge \psi, \vec{d} : D) =& b \wedge Cluster_\vee(\psi, \vec{d} : D) \\
Cluster_\vee(\psi_1 \wedge \psi_2, \vec{d} : D) =& c_*(\vec{d}) \\
& \textit{where } (c_*(\vec{d} : D) = Cluster_\wedge(\psi_1 \wedge \psi_2, \vec{d} : D)) \textit{ is a fresh cluster} \\
Cluster_\vee(\psi_1 \vee \psi_2, \vec{d} : D) =& Cluster_\vee(\psi_1, \vec{d} : D) \vee Cluster_\vee(\psi_2, \vec{d} : D) \\
Cluster_\vee(\forall \vec{v} : D'.\psi, \vec{d} : D) =& c_*(\vec{d}) \\
& \textit{where } (c_*(\vec{d} : D) = Cluster_\wedge(\forall \vec{v} : D'.\psi)) \textit{ is a fresh cluster} \\
Cluster_\vee(\exists \vec{v} : D'.\psi, \vec{d} : D) =& \exists \vec{v} : D'.Cluster_\vee(\psi, (\vec{d} +\!\!+ \vec{v} : D \times D'))
\end{aligned}
$$

The second parameter to the Cluster functions represents the (typed) parameter list. It is accumulated in order to describe the parameter lists of fresh clusters. Note that c_* represent fresh cluster names. By convention, we replace the $_*$ by an unused natural number.

The fact that the *Cluster* functions on PBESs in GNF emit clustered PBESs complying with Definition 3.2 is easily proved by showing that *Cluster* transforms a GNF equation into the form $CLUSTERED_EQ$ and $Cluster_\wedge$ and $Cluster_\vee$ produce new clusters of the form $CLUSTER$. Note that apart from splitting of subformulas, the *Cluster* functions describe simple traversals of the syntax tree of GNF, making it easy to see that the transformation is sound and well-defined.

Example 11 (Cluster identification) *Assume the following PBES equation:*

$$\nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee (\exists d : \mathbb{B}.true \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c))))$$

Note that this equation is in GNF. This equation was constructed such that it contains a symbol switch (from conjunctive to disjunctive), causing introduction of a cluster.

Clusters can be identified in this formula as follows:

$$
\begin{aligned}
& Cluster(\nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee (\exists d : \mathbb{B}.true \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c))))) \\
=& \nu X(b : \mathbb{B}, c : \mathbb{B}) = Cluster(\neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee (\exists d : \mathbb{B}.true \\
& \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c)))), (b : \mathbb{B}, c : \mathbb{B})) \\
=& \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow Cluster(X(b \vee c, \neg c) \vee (\exists d : \mathbb{B}.true \\
& \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c))), (b : \mathbb{B}, c : \mathbb{B})) \\
=& \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow Cluster_\vee(X(b \vee c, \neg c) \vee (\exists d : \mathbb{B}.true \\
& \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c))), (b : \mathbb{B}, c : \mathbb{B})) \\
=& \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (Cluster_\vee(X(b \vee c, \neg c), (b : \mathbb{B}, c : \mathbb{B})) \vee Cluster_\vee(\exists d : \mathbb{B}.true
\end{aligned}
$$

$$
\begin{aligned}
&\quad \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c)), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}))) \\
&= \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d : \mathbb{B}.\mathit{Cluster}_{\vee}(\mathit{true} \\
&\quad \wedge (X(b \wedge c, d) \wedge X(b \wedge d, c)), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}))) \\
&= \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d : \mathbb{B}.\mathit{true} \\
&\quad \wedge \mathit{Cluster}_{\vee}(X(b \wedge c, d) \wedge X(b \wedge d, c), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}))) \\
&= \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d : \mathbb{B}.\mathit{true} \wedge c_1(b, c, d)) \\
&\quad c_1(b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}) = \mathit{Cluster}_{\wedge}(X(b \wedge c, d) \wedge X(b \wedge d, c), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B})) \\
&= \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d : \mathbb{B}.\mathit{true} \wedge c_1(b, c, d)) \\
&\quad c_1(b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}) = \mathit{Cluster}_{\wedge}(X(b \wedge c, d), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B})) \\
&\quad \wedge \mathit{Cluster}_{\wedge}(X(b \wedge d, c), (b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B})) \\
&= \nu X(b : \mathbb{B}, c : \mathbb{B}) = \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d : \mathbb{B}.\mathit{true} \wedge c_1(b, c, d)) \\
&\quad c_1(b : \mathbb{B}, c : \mathbb{B}, d : \mathbb{B}) = X(b \wedge c, d) \wedge X(b \wedge d, c)
\end{aligned}
$$

Because of the way the *Cluster* functions work, it is apparent that the strength of the guards from the original PBES in GNF is retained. Because these guards are not placed in the top level of a cluster, but rather as a guard for the references to the cluster, these guards directly determine whether the cluster must be instantiated. Using strongly guarded GNF as input, we find a clustered PBES in which all reachable instantiations of clusters contain at least one recursion variable. If this was not the case, then at least one of the guards in the cluster evaluated to false, while all the guards of the cluster itself evaluated to true. Because the guards of the cluster are strengthened by the guards inside the cluster, this can never happen.

Example 12 (No unnecessary infinite instantiations) *Recall the PBES from Example 9, which is trivially translated to GNF by changing the disjunction into an implication. When using the Cluster functions on it directly, we would obtain the results from this example, containing the infinite instantiation. By first transforming the system to SGF, as done in Example 10, this is no longer the case:*

$$
\begin{aligned}
&\mathit{Cluster}(\nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.(i \leqslant 10) \Rightarrow \exists j : \mathbb{N}.((i > 10) \vee (j < 10)) \wedge (i \leqslant 10) \\
&\quad \Rightarrow ((j < 10) \wedge X(i, j)))) \\
&= \nu X(n : \mathbb{N}, m : \mathbb{N}) = (\forall i : \mathbb{N}.(i \leqslant 10) \Rightarrow c_1(n, m, i)) \\
&\quad c_1(n : \mathbb{N}, m : \mathbb{N}, i : \mathbb{N}) = (\exists j : \mathbb{N}.((i > 10) \vee (j < 10)) \wedge (i \leqslant 10) \Rightarrow ((j < 10) \wedge X(i, j)))
\end{aligned}
$$

Because c_1 is guarded by the expression $i \leqslant 10$, there are no more infinite instantiations.

Depending on the used method of instantiation, another type of unnecessary infinite instantiations can occur. Assume equation $\nu X = \exists n : \mathbb{N}.(X \wedge (n \neq 0 \vee Y(m)))$ where $\nu Y(n : \mathbb{N}) = \phi$ is some equation. Instantiation results in the BES equation $\nu X = (X \wedge Y_0) \vee \bigvee_{n \in \mathbb{N}} X$, which is trivially equal to $\nu X = (X \wedge Y_0) \vee X$. A smart instantiation procedure will recognize this and instantiate to this last BES equation.

After clustering however, instantiation yields $\nu X = c_0 \vee c_1 \vee ...$ where $c_n = X$ for all $n \in \mathbb{N}$ if $n > 0$. Now the same optimization can not be used as easily, because it would require parameter elimination in clusters. These situations can often be prevented by careful preprocessing (because X is independent of m, it can be moved outside of the existential quantification).

4 Exploration

Instantiating a clustered PBES corresponds to generating a structure graph. Because the instantiation of a cluster or PBES equation in a clustered PBES yields a purely conjunctive or disjunctive BES equation, a single instantiation of a cluster or equation corresponds to a single node and its outgoing edges in the structure graph. We denote a node corresponding to the instantiation of an equation $\mathcal{X}$ with parameters $\vec{d}$ in the context of equation system $\mathcal{E}$ by $\left\langle \mathcal{X}, \vec{d}, \mathcal{E} \right\rangle$. For simplicity, we define a function *name* such that $name(\left\langle \mathcal{X}, \vec{d}, \mathcal{E} \right\rangle) = \mathcal{X}$.

We use state-space exploration to find all nodes and transitions of the structure graph that are reachable from the initial state of the PBES. As a first step, we define the next-state function, which is used to explore the structure graph over the edge relation. Based on this function, an algorithm is presented to perform the instantiation explicitly. After this, symbolic exploration techniques are described and applied on the exploration of structure graphs.

4.1 Next-state function

In general, exploration of a state-space is based on a next-state function, which determines for any state its complete set of outgoing edges, exploring the transition relation. The next-state function in the case of exploring a structure graph corresponds to the instantiation of an equation of a clustered PBES.

We define function $nextState_{\mathcal{E}}$ as the next-state function for the exploration of the structure graph corresponding to a system $\mathcal{E}$. For all states $s \in S$ in a structure graph $\mathcal{G} = \langle S, s_0, \rightarrow, d, r \rangle$, it holds that $nextState_{\mathcal{E}}(s) = \{t \mid (s,t) \in \rightarrow\}$.

Definition 4.1 (Function $nextState_{\mathcal{E}}$) *Assume a PBES $\mathcal{E}$ such that $\mathcal{E}$ contains an equation $(\sigma\mathcal{X}(\vec{d}) = \phi)$ and assume a structure graph node $s = \langle \mathcal{X}, \vec{v}, \mathcal{E} \rangle$.*

Then $nextState_{\mathcal{E}}(s)$ is equal to $nextStateF(\phi)(\varepsilon[\vec{d} := \vec{v}])$ for some data environment ε, where function nextStateF is inductively defined as follows:

$$
\begin{aligned}
nextStateF(b)\varepsilon &= \begin{cases} \{\langle \top, \langle\rangle, \mathcal{E} \rangle\} & \text{if } [\![b]\!]\varepsilon \\ \{\langle \bot, \langle\rangle, \mathcal{E} \rangle\} & \text{otherwise} \end{cases} \\
nextStateF(\mathcal{X}(e))\varepsilon &= \{\langle \mathcal{X}, [\![e]\!]\varepsilon, \mathcal{E} \rangle\} \\
nextStateF(c(e))\varepsilon &= \{\langle c, [\![e]\!]\varepsilon, \mathcal{E} \rangle\} \\
nextStateF(b \Rightarrow \psi)\varepsilon &= \begin{cases} nextStateF(\psi)\varepsilon & \text{if } [\![b]\!]\varepsilon \\ \{\langle \top, \langle\rangle, \mathcal{E} \rangle\} & \text{otherwise} \end{cases} \\
nextStateF(b \wedge \psi)\varepsilon &= \begin{cases} nextStateF(\psi)\varepsilon & \text{if } [\![b]\!]\varepsilon \\ \{\langle \bot, \langle\rangle, \mathcal{E} \rangle\} & \text{otherwise} \end{cases} \\
nextStateF(\psi_1 \wedge \psi_2)\varepsilon &= nextStateF(\psi_1)\varepsilon \cup nextStateF(\psi_2)\varepsilon \\
nextStateF(\psi_1 \vee \psi_2)\varepsilon &= nextStateF(\psi_1)\varepsilon \cup nextStateF(\psi_2)\varepsilon \\
nextStateF(\exists \vec{v} : D'.\psi)\varepsilon &= \bigcup_{\vec{w} \in D'} nextStateF(\psi)(\varepsilon[\vec{v} := \vec{w}]) \\
nextStateF(\forall \vec{v} : D'.\psi)\varepsilon &= \bigcup_{\vec{w} \in D'} nextStateF(\psi)(\varepsilon[\vec{v} := \vec{w}])
\end{aligned}
$$

The same rules can be used on a node $s' = \langle c, \vec{v}, \mathcal{E} \rangle$ such that $\mathcal{E}$ contains a cluster $c(\vec{d}) = \phi$. We introduce nodes $\langle \top, \langle\rangle, \mathcal{E} \rangle$ and $\langle \bot, \langle\rangle, \mathcal{E} \rangle$ as special, unranked nodes such that $d(\langle \bot, \langle\rangle, \mathcal{E} \rangle) = \top$ and $d(\langle \bot, \langle\rangle, \mathcal{E} \rangle) = \bot$. Furthermore $nextState_{\mathcal{E}}(\langle \bot, \langle\rangle, \mathcal{E} \rangle) = nextState_{\mathcal{E}}(\langle \top, \langle\rangle, \mathcal{E} \rangle) = \emptyset$.

4.2 Explicit Exploration

The $nextState_{\mathcal{E}}$ function is used to explore the transition relation ($\rightarrow$). When exploring structure graphs however, this is not sufficient for finding a complete structure graph. The correct values for the d and r mappings must also be found.

The ranking mapping r is simple to define. For all clusters ($c(\vec{d} : D) = \phi$) with parameters $\vec{v}$ in the context of an equation system $\mathcal{E}$, $r(\langle c, \vec{v}, \mathcal{E}\rangle)$ is undefined. For all fixed-point equations ($\sigma\mathcal{X}(\vec{d'} : D') = \phi'$) with parameters $\vec{v}'$, it holds that $r(\langle \mathcal{X}, \vec{v}', \mathcal{E}\rangle) = rank_{\mathcal{E}}(\mathcal{X})$. Note that the value for r only depends on the name of the recursion variable, and can thus be computed using only that name.

The decoration mapping d depends on the name of the recursion variable as well. It can be seen that *nextStateF* always generates a non-empty set of states, and thus the case in which there are no next states does not need to be evaluated. If the number of next states to s is 1, then following the SOS rules from Definition 2.5, $d(s)$ may be undefined. However, we always assign it either $\wedge$ or $\vee$, depending on the recursion variable or cluster the state corresponds to. This will not invalidate the BESsyness property, described in Section 2.1.1. We assign $d(s)$ the symbol $\wedge$ if s corresponds to a conjunctive equation or conjunctive cluster, and $\vee$ otherwise. Although it can be determined whether an equation or cluster is conjunctive or disjunctive after its generation by matching grammars, it is easier to do this during the clustering operation, by denoting whether $Cluster_{\wedge}$ or $Cluster_{\vee}$ was used in the cluster. Note that if both grammars match (and consequently, neither transformation function was used), the number of next states to s can not exceed 1. If the clustered PBES was built using the *Cluster* functions, this only occurs in equations (as opposed to clusters) and the node to which the single transition refers is either a recursion variable or the $true$- or $false$-node. In that case, both decoration $\wedge$ and $\vee$ are allowed.

Algorithm 1 describes the process of explicitly instantiating the state-space of a clustered PBES. In order to instantiate the structure graph of a system $\mathcal{E}$ with initial state $X(\vec{d})$ for some parameter vector $\vec{d}$, we can call EXPLORE-PBES $\left(\left\langle X, \vec{d}, \mathcal{E}\right\rangle\right)$.

This algorithm takes an initial state and generates the state space as a BES structure graph using the $nextState_{\mathcal{E}}$ function to explore the transition relation. The complete structure graph is returned.

4.3 Symbolic Exploration

The explicit exploration of a structure graph, described in Section 4.2, can be combined with existing symbolic exploration techniques ([11, 12]) in order to improve both the performance of the algorithm and yield a more compact representation of the algorithm's results. In order to efficiently use these symbolic techniques, we first define partitioned transition relations in the scope of structure graph exploration.

4.3.1 Partitioned Transition Relation

The fact that the result of the instantiation of a cluster or equation in a clustered PBES yields a fully conjunctive or disjunctive BES equation can be exploited by using *partitioned transition functions.* Partitioned transition functions were introduced in [8], in the scope of symbolic model checking. Instead of generating a single monolithic transition relation ($nextState_{\mathcal{E}}$), it is possible to generate several partial ones. The combination of these partial transition relations results in the complete transition relation.

Definition 4.2 ($nextState_{\mathcal{E}\mathcal{X}}$) *We define function* $nextState_{\mathcal{E}\mathcal{X}}(s)$ *as follows:*

$$nextState_{\mathcal{E}\mathcal{X}}(s) = \left\{s' \in S | s \rightarrow s' \wedge \exists \vec{d}. s = \left\langle \mathcal{X}, \vec{d}, \mathcal{E}\right\rangle\right\}$$

Algorithm 1 EXPLORE-PBES

```
1: function EXPLORE-PBES(s_0)
2:     Ss := {s_0}                                              ▷ Unexplored state vectors
3:     S := {⟨⊤, ⟨⟩, ℰ⟩, ⟨⊥, ⟨⟩, ℰ⟩}                            ▷ Explored state vectors
4:     → := ∅                                                   ▷ Explored transition relation
5:     d := ∅                                                   ▷ Explored decoration mapping
6:     d[⟨⊤, ⟨⟩, ℰ⟩] := ⊤
7:     d[⟨⊥, ⟨⟩, ℰ⟩] := ⊥
8:     r := ∅                                                   ▷ Explored ranking mapping
9:     while Ss ≠ ∅ do
10:        s := some element of Ss
11:        T := nextState_ℰ(s)                                  ▷ Explore the transition relation
12:        Ss := Ss ∪ {t | t ∈ T ∧ t ∉ S}
13:        → := → ∪ {⟨s, t⟩ | t ∈ T}
14:        Ss := Ss − {s}
15:        S := S ∪ {s}
16:        d[s] := { ∧  if name(s) is a conjunctive cluster or equation
                   { ∨  otherwise
17:        if name(s) is not the name of a cluster then         ▷ Find the rank
18:            r[s] := rank_ℰ(𝒳)
19:        end if
20:    end while
21:    return ⟨S, s_0, →, d, r⟩
22: end function
```

This function describes for any state s corresponding to cluster or recursion variable $\mathcal{X}$ in the context of a system $\mathcal{E}$ the complete set of target states.

It trivially holds that $\mathit{nextState}_{\mathcal{E}}$ is the union of $\mathit{nextState}_{\mathcal{E}\mathcal{X}}$ for all cluster names and recursion variables $\mathcal{X}$. It can also be seen that the functions $\mathit{nextState}_{\mathcal{E}\mathcal{X}}$ can easily be found using function *nextStateF* from Definition 4.1.

Example 13 (Partitioned transition relation) *Assume the following (clustered) PBES $\mathcal{E}$:*

$$\nu X(n : \mathbb{N}, b : \mathbb{B}) =(n > 0) \Rightarrow (X(n, \neg b) \vee Y(n, b))$$
$$\nu Y(n : \mathbb{N}, b : \mathbb{B}) =X(n - 1, b)$$

The value of $\mathit{nextState}_{\mathcal{E}X}(s)$ for state $s = \langle X, \langle n, b\rangle, \mathcal{E}\rangle$ with $n \in \mathbb{N}$ and $b \in \mathbb{B}$ can be denoted as $\{\langle X, \langle n, \neg b\rangle, \mathcal{E}\rangle \mid n > 0\} \cup \{\langle Y, \langle n, b\rangle, \mathcal{E}\rangle \mid n > 0\} \cup \{\langle \top, \langle\rangle, \mathcal{E}\rangle \mid \neg(n > 0)\}$, following the structure of nextStateF. In this notation, the partial transition function $\mathit{nextState}_{\mathcal{E}\mathcal{X}}$ is partitioned even further. All possible targets for the transition are included, and the expressions guarding the inclusion of the target in the PBES also guard the inclusion of the target here.

Assume $s = \langle X, \langle n, b\rangle, \mathcal{E}\rangle$. We denote the partitions as follows:

$$\mathit{nextState}_{\mathcal{E}X,1}(s) =\{\langle X, \langle n, \neg b\rangle, \mathcal{E}\rangle \mid n > 0\} \cup \{\langle \top, \langle\rangle, \mathcal{E}\rangle \mid \neg(n > 0)\}$$
$$\mathit{nextState}_{\mathcal{E}X,2}(s) =\{\langle Y, \langle n, b\rangle, \mathcal{E}\rangle \mid n > 0\} \cup \{\langle \top, \langle\rangle, \mathcal{E}\rangle \mid \neg(n > 0)\}$$
$$\mathit{nextState}_{\mathcal{E}X}(s) =\mathit{nextState}_{\mathcal{E}X,1}(s) \cup \mathit{nextState}_{\mathcal{E}X,2}(s)$$

It is possible to calculate the $\mathit{nextState}_{\mathcal{E}X,*}$ functions separately and then combine them to find $\mathit{nextState}_{\mathcal{E}X}$, thereby exploring the transition relation for X.

We use *transition groups* in order to define the partitioned transition relation. We split an equation or cluster by reference. For example, the equation $\nu X(n : \mathbb{N}, b : \mathbb{B}) = (n > 0) \Rightarrow$

$X(n, \neg b) \vee Y(n, b)$ has two references to equations. We split this equation in transition groups by creating a group containing $X(n, \neg b)$ and its guard $(n > 0)$, and another group containing $Y(n, b)$ and again $(n > 0)$, which also guards this subexpression. In addition to simplifying the exploration, the use of transition groups helps when using techniques exploiting *event locality*. Event locality is presented in Section 4.4.

Instead of having a single reference per group, it is possible to have multiple by merging groups. Doing this may reduce the amount of relations that need to be explored, possibly resulting in more efficient algorithms. This merging is described in more detail in Section 4.5.

4.4 Event Locality

Event locality describes the fact that often in a transition system, events do not influence the whole system but just a part of it. For example, the event in which a switch in a complex control system is toggled may only change the value of a parameter, leaving the rest of the system untouched. This event locality shows up in the data parameters of the processes in the process specification.

In the field of model checking, PBESs usually follow from the combination of a process specification and a property expressed in modal logic, through a process described in [15]. This transformation retains the event locality as expressed by data parameters in the process specification. For this reason, PBESs generated this way generally have the same property.

The following system demonstrates event locality in a PBES:

$$\begin{aligned}\nu X(n:\mathbb{N}, b:\mathbb{B}) =& X(n, \neg b) \vee Y(n, b)\\ \nu Y(n:\mathbb{N}, b:\mathbb{B}) =& X(n+1, b)\end{aligned}$$

It can be seen that X refers to itself and Y, twice with an unchanged parameter n, while Y refers only to X with an unchanged parameter b. The fact that not all parameters change in these references corresponds with the locality of events.

In [3], it is shown that event locality can be exploited in order to reduce the number of computations that have to be done in a state space exploration. In [20] this is used for efficient instantiation of PBESs to parity games. In particular, the transition relations in parity games (which correspond to references in BESs) are both explored and represented more efficiently.

In state-space exploration, event locality can be exploited by using local transition relations. This may greatly reduce the amount of calls to $nextState_{\mathcal{E}}$ for instances in which only a few parameters are changed. Fewer $nextState_{\mathcal{E}}$ calls leads to performance improvements, because the $nextState_{\mathcal{E}}$ function may involve complex rewrite steps, whereas applying an existing local transition is a very simple and fast operation. Section 4.4.2 further illustrates this.

4.4.1 Dependency

We define a notion of dependency in clustered PBESs based on event locality. In order to do this, a few functions are defined.

Definition 4.3 (*changed*) *Function changed gives the indices of the elements of the parameter vector that may be changed by a formula. For a data expression described by the function* $e : (D_1 \times D_2 \times ... \times D_m) \rightarrow (E_1 \times E_2 \times ... \times E_n)$*, the following holds:*

$$\begin{aligned} changed(e) = & \Big\{ i \mid i \leqslant m \wedge i \leqslant n \wedge \exists \varepsilon . [\![\vec{d_i}]\!]\varepsilon \neq [\![e(\vec{d})_i]\!]\varepsilon \Big\} \\ & \cup \{ i \mid m < i \leqslant n \} \end{aligned}$$

If the size of the output vectors is greater than the size of the input vectors, the newly introduced elements are always in changed, due to the second part of the definition. If the result vectors of e have a smaller length than the input vectors, the removed indices are not in changed, unless they occur in another subexpression.

Definition 4.4 (*changed* equation) *Consider an equation or cluster $\mathcal{X}$.*

We define changed($\mathcal{X}$) as the union of changed(e) for all occurring data expressions e, limited by the length of the parameter vector to $\mathcal{X}$. Note that such data expressions only occur as parameters to recursion variables and cluster references, by Definition 3.1.

Definition 4.5 (*used*) *Function used gives the parameters that occur somewhere in the formula, and are not merely passed to the following state.*

For a data expression described by the function $e(\vec{d}) = \langle e_1, e_2, ..., e_n \rangle$, we define the following:

$$\begin{aligned} used(e) =& \{i \mid \exists j. d_i \in FV(e_j) \wedge 1 \leqslant i \leqslant n \wedge 1 \leqslant j \leqslant n \wedge i \neq j\} \\ & \cup \{i \mid d_i \in FV(e_i) \wedge i \in changed(e) \wedge i \leqslant n\} \end{aligned}$$

Where $FV(e)$ is the set of free variables in e.

The first part of this definition handles the variables that occur in expressions for different indices than their own. If this happens, the variable is always in *used*. The second part of the definition handles the case where the variable occurs in the expression for its own index, but is part of a more complex expression, possibly involving other variables.

For example, assume an expression described by the function $e(n : \mathbb{N}, m : \mathbb{N}, o : \mathbb{N}) = \langle n, m{+}n, o \rangle$. Parameter n (index 1) is in $used(e)$, because it occurs in the expression $m + n$ and is therefore not merely passed to the next state. Parameter m (index 2) is in $used(e)$ for the same reason. Parameter o (index 3) is not in $used(e)$, because it is passed to the next state unchanged and does not occur in the expressions for the other indices.

Definition 4.6 (*used* equation) *Assume an equation $\mathcal{X}$ with parameters $\vec{d}$ and predicate formula ϕ.*

We define used($\mathcal{X}$) as the union of used(e) for all occurring data expressions e and the indices of all parameters occurring in simple boolean expressions (guards). Thus, $used(\mathcal{X}) = used(\phi)$ where $used(\phi)$ is defined recursively as follows:

$$\begin{aligned} used(\mathcal{X}'(e)) =& used(e) \cap \{i \mid 1 \leqslant i \leqslant |\vec{d}|\} \\ used(c(e)) =& used(e) \cap \{i \mid 1 \leqslant i \leqslant |\vec{d}|\} \\ used(b \Rightarrow \psi) =& \{i \mid d_i \in FV(b)\} \cup used(\psi) \\ used(b \wedge \psi) =& \{i \mid d_i \in FV(b)\} \cup used(\psi) \\ used(\forall \vec{v} : D.\psi) =& used(\psi) \\ used(\exists \vec{v} : D.\psi) =& used(\psi) \end{aligned}$$

The same definition can be used if $\mathcal{X}$ is not a recursion variable, but a cluster name.

Note that we only allow parameter indices (as opposed to variables bound by a quantifier) to be contained in the result of *used*.

Example 14 (*changed* and *used*) *Assume X defined as follows:*

$$\nu X(b : \mathbb{B}, c : \mathbb{B}) = X(b \vee c, \neg c) \vee (\exists d \in \mathbb{B}. true \wedge c_1(b, c, d))$$

We calculate changed(X) and used(X) as follows:

$$\begin{aligned} changed(X) =& changed(\lambda \langle b, c \rangle : \mathbb{B} \times \mathbb{B}. \langle b \vee c, \neg c \rangle) \cup changed(\lambda \langle b, c, d \rangle : \mathbb{B} \times \mathbb{B} \times \mathbb{B}. \langle b, c, d \rangle) \\ =& \{1, 2\} \cup \emptyset \\ =& \{1, 2\} \\ used(X) =& used(\lambda \langle b, c \rangle : \mathbb{B} \times \mathbb{B}. \langle b \vee c, \neg c \rangle) \cup used(\lambda \langle b, c, d \rangle : \mathbb{B} \times \mathbb{B} \times \mathbb{B}. \langle b, c, d \rangle) \cup \emptyset \\ =& \{1, 2\} \cup \{1, 2\} \cup \emptyset \cup \emptyset \\ =& \{1, 2\} \end{aligned}$$

Definition 4.7 (*changed* **and** *used* **in partitioned transitions)** *Assume an equation $\mathcal{X}$ with n transition groups. We define changed$(\mathcal{X}, i)$ as changed(ϕ) and used$(\mathcal{X}, i)$ as changed(ϕ) where $\mathcal{X}^i = \phi$ is the equation describing transition group i (with $1 \leqslant i \leqslant n$).*

Because the set *changed*(e) is in general not computable, we may approximate it syntactically. An expression e can be written in the form $\langle e_0, e_1, ..., e_n \rangle$. We approximate *changed*$^{\approx}(e)$ as the set of i for which $e_i \neq d_i$ (e_i is not described by just the symbol d_i) or $m \leqslant i < n$. This also leads to a syntactic approximation of *used*.

For example, if $e(n : \mathbb{N}, m : \mathbb{N}, o : \mathbb{N}) = \langle n, m+n, 2o-o \rangle$, then *changed*$^{\approx}(e) = \{2, 3\}$ (parameters m and o), because the second parameter in the results $(m + n)$ is not equal to the symbol m and the third parameter $(2o - o)$ is not equal to the symbol o, although the expressions are equivalent.

If we assume a data expression described by the function $e(n : \mathbb{N}, m : \mathbb{N}) = \langle n, m + n, n \rangle$, then *changed*$^{\approx}(e) = \{2, 3\}$ (parameters m and the newly introduced one).

In the remainder of this document, *changed* and *used* may be replaced with *changed*$^{\approx}$ and *used*$^{\approx}$, for practical purposes.

The functions *changed* and *used* describe the following two types of dependencies:

- Read dependency

  There is a read dependency of an equation or cluster $\mathcal{X}$ on a variable d_x if the variable is used in any of the data expressions, or in a guard. This is true if and only if $x \in$ *used*$(\mathcal{X})$.

- Write dependency

  There is a write dependency of an equation or cluster $\mathcal{X}$ on a variable d_x if the value for the parameter is changed while passing it on. This holds if and only if $x \in$ *changed*$(\mathcal{X})$.

When looking at dependencies based on event locality, we only evaluate the parameter vectors. Although such a dependency could also be found between names of equations or clusters, it would not be very useful. The reason for this is that read dependency on the name would be required to always be true (since the equation itself depends on it) and since recursion variable names can not be a function of parameters, symbolic exploration will not benefit from exploiting write dependencies.

4.4.2 Local transition relation

Using the dependency information from Section 4.4.1, local transitions can be defined. Local transitions describe a transition of only the part of a state which changes as a result of the transition.

We introduce local transitions by means of an example.

Assume the following PBES $\mathcal{E}$:

$$\nu X(n : \mathbb{N}, m : \mathbb{N}) = Y(f(n), m)$$
$$\nu Y(n : \mathbb{N}, m : \mathbb{N}) = n < 10 \wedge X(n, f(n))$$

Here, f is a function of type $\mathbb{N} \to \mathbb{N}$. For simplicity, we assume $f(0) = 1$.

Clearly, *used*$(X) = \{1\}$, *changed*$(X) = \{1\}$, *used*$(Y) = \{1\}$ and *changed*$(Y) = \{2\}$.

When exploring the transition relation from any state $\langle X, \langle 0, m \rangle, \mathcal{E} \rangle$ with $m \in \mathbb{N}$, we find the transition $\langle X, \langle 0, m \rangle, \mathcal{E} \rangle \to \langle Y, \langle 1, m \rangle, \mathcal{E} \rangle$. Because the the transition function for X is not read- or write dependent on the second parameter to X, the value of this parameter can simply be copied in all cases. We describe this by saying that the transition function is local to only the first parameter. When exploring any state $\langle X, \langle 0, m \rangle, \mathcal{E} \rangle$ for any $m \in \mathbb{N}$, this transition function can be used to quickly determine the next state. After calculating this local transition function once, it is not necessary to recompute the value of $f(0)$. This is particularly useful when this involves complex calculations.

There exist cases in which a transition is only read dependent or write dependent on a parameter. An example of this is seen in equation Y of $\mathcal{E}$. This equation is read dependent on the first parameter and write dependent on the second parameter. In such a case, the new values of the write depended parameters depend only on the values of the read depended parameters. Assuming a state $\langle Y, \langle 0, m \rangle, \mathcal{E} \rangle$ for some $m \in \mathbb{N}$ will yield the symbolic transition relation $\langle Y, \langle 0, m \rangle, \mathcal{E} \rangle \rightarrow \langle X, \langle n, 1 \rangle, \mathcal{E} \rangle$, where the value of n in the target state is always the same as the first parameter in the source state.

We use the symbol $*$ to denote elements of the transition which are not relevant to the event corresponding to the transition. A $*$ symbol in the source state denotes any value, while in a target state it denotes the same value as the parameter in the source state with the same index. We can denote the transition from $\langle X, \langle 0, m \rangle, \mathcal{E} \rangle$ as $\langle X, \langle 0, * \rangle, \mathcal{E} \rangle \rightarrow \langle Y, \langle 1, * \rangle, \mathcal{E} \rangle$ and the transition from $\langle Y, \langle 0, m \rangle, \mathcal{E} \rangle$ as $\langle Y, \langle 0, * \rangle, \mathcal{E} \rangle \rightarrow \langle X, \langle *, 1 \rangle, \mathcal{E} \rangle$. By using this notation in our transition representation, the amount of transitions that must be denoted to represent an entire state-space can be much smaller.

Example 15 (Local transitions) *Assume PBES $\mathcal{E}$ from Example 13:*

$$\begin{aligned} \nu X(n : \mathbb{N}, b : \mathbb{B}) &= (n > 0) \Rightarrow (X(n, \neg b) \vee Y(n, b)) \\ \nu Y(n : \mathbb{N}, b : \mathbb{B}) &= X(n - 1, b) \end{aligned}$$

In this PBES, it holds that $changed(X) = \{2\}$ and $used(X) = \{1, 2\}$.

The local transitions from state $\langle X, \langle 1, false \rangle, \mathcal{E} \rangle$ can be found by calculating the next-states $nextState_{\mathcal{E}}(\langle X, \langle 1, false \rangle, \mathcal{E} \rangle)$, which results in the following:

$$\{\langle X, \langle 1, true \rangle, \mathcal{E} \rangle, \langle Y, \langle 1, false \rangle, \mathcal{E} \rangle\}$$

We can then overlay these states with $*$ symbols where there is no write dependency:

$$\{\langle X, \langle *, true \rangle, \mathcal{E} \rangle, \langle Y, \langle *, false \rangle, \mathcal{E} \rangle\}$$

Because there is a read dependency on all parameters, the source state remains unchanged. The result is the following set of transitions:

$$\left\{ \begin{array}{c} \langle X, \langle 1, false \rangle, \mathcal{E} \rangle \rightarrow \langle X, \langle *, true \rangle, \mathcal{E} \rangle \\ \langle X, \langle 1, false \rangle, \mathcal{E} \rangle \rightarrow \langle Y, \langle *, false \rangle, \mathcal{E} \rangle \end{array} \right\}$$

Note that while the dependencies for the first transition group are the same ($changed(X, 1) = \{2\}$ and $used(X, 1) = \{1, 2\}$), for the second transition group, there are fewer dependencies ($changed(X, 2) = \emptyset$ and $used(X, 2) = \{1\}$).

When looking at the second transition group, the source state can be rewritten to $\langle X, \langle 1, * \rangle, \mathcal{E} \rangle$ and the target state to $\langle Y, \langle *, * \rangle, \mathcal{E} \rangle$, generalizing the transition further, removing the need to store the transition of the second group for all values of b.

When using a partitioned relation, we find the following set of transitions:

$$\left\{ \begin{array}{r} \langle X, \langle 1, false \rangle, \mathcal{E} \rangle \rightarrow \langle X, \langle *, true \rangle, \mathcal{E} \rangle \\ \langle X, \langle 1, * \rangle, \mathcal{E} \rangle \rightarrow \langle Y, \langle *, * \rangle, \mathcal{E} \rangle \end{array} \right\}$$

4.5 Merging Groups

Instead of creating a single transition group per reference to a recursion variable or cluster, similar transition groups can be combined.

Merging transitions groups is only useful if the two merged groups have similar dependencies.

4.6 Linked Decision Diagrams

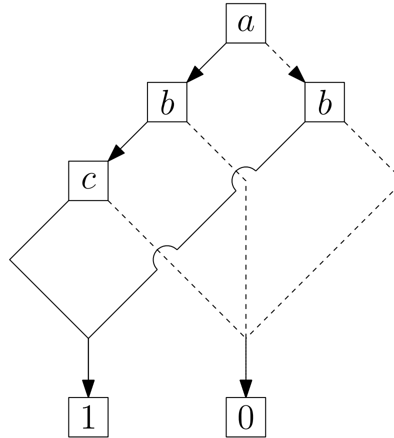
Linked decision diagrams (LDDs) are the symbolic data structure that we propose to use in the exploration. In order to define LDDs, we first define binary decision diagrams.

Binary decision diagrams (BDDs) are data structures that are used for compact symbolic storage of boolean functions and manipulation thereof ([7, 6]).

BDDs are directed acyclic graphs (DAGs) in which each node represents a boolean property or variable, except for the two final nodes 0 and 1. All nodes except these special nodes have two outgoing edges, where one edge represents $true$ as a value for the property or variable and the other one represents $false$. Nodes 0 and 1 represent the outcome of the formula represented by the BDD ($false$ or $true$ respectively). A BDD contains an initial node, which has no incoming edges. Apart from the initial node and nodes 0 and 1, all nodes must have at least one incoming edge. By enumerating all paths from the initial node to 1, a set of all possible combinations of values for the properties or variables is found, such that the result of the boolean function represented by the BDD is $true$.

We make use of Ordered BDDs (OBDDs). This means that a total ordering $\lessdot$ exists on all variables in the graph. A node corresponding to a certain variable a may only have an edge to a node corresponding to variable b if $a \lessdot b$.

Example 16 (BDD) *The following BDD is a representation of the formula* $((\neg a \wedge b) \vee (b \wedge c))$.



The normal edges represent true and the dashed edges represent $false$. *The ordering* $a \lessdot b \lessdot c$ *is used.*

Following all paths from the initial node ("a") to node 1 *yields the set of valuations such that the represented expression* $((\neg a \wedge b) \vee (b \wedge c))$ *holds:*

$$\left\{ \begin{array}{c} \left[\begin{array}{c} a := true \\ b := true \\ c := true \end{array} \right] \\ \\ \left[\begin{array}{c} a := false \\ b := true \end{array} \right] \end{array} \right\}$$

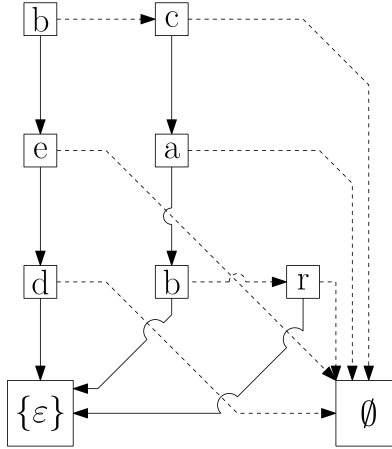
Linked decision diagrams (LDDs), presented first in [2], use a method that is similar to BDDs in order to store sets of strings or vectors. In LDDs, node 1 is called $\{\varepsilon\}$, node 0 is called $\emptyset$ and all other nodes have a label.

All labeled nodes n in an LDD have two outgoing edges: $n \rightarrow n_1$ (*"positive"*, often represented by an arrow going downward from the node) and $n \rightarrow n_2$ (*"negative"* often represented by an arrow starting at the right of the node).

Definition 4.8 (LDD semantics) *Assume an LDD node n with positive edge $n \rightarrow n_1$ and negative edge $n \rightarrow n_2$. Each subDAG (node and its transitive successors) of such a node n represents a set of strings as follows:*

$$\begin{aligned} [\![\{\epsilon\}]\!] &= \{\epsilon\} \\ [\![\emptyset]\!] &= \emptyset \\ [\![a]\!] &= \{aw \mid w \in [\![n_1]\!]\} \cup [\![n_2]\!] \end{aligned}$$

Example 17 (LDD) *An LDD for the set of strings $\{bed, cab, car\}$ is given in the following figure:*

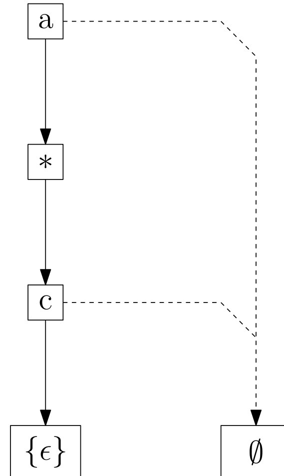


The negative edges are shown dashed. Note that the c and a symbols, which occur twice in the set of strings, occur only once in the LDD.

Extended LDDs LDDs can only represent finite sets, because all included values for all elements are enumerated. When representing transitions, it becomes useful to allow a vector element to have any value. In order to represent this, we extend LDDs with a unary $*$ ("wildcard") node, and we extend the semantics as given in Definition 4.8 by the semantics for a $*$ node with a singleton edge to a node n:

$$[\![*]\!] = \{xw | x \in U \wedge w \in [\![n]\!]\}$$

Example 18 (Extended LDDs) *Using extended LDDs, we can represent the set $\{v \mid v_1 = a \wedge v_3 = c\}$. The LDD representing this set is given in the following figure:*



Using LDDs, it is possible to share common parts in the strings. This makes LDDs particularly useful for efficiently storing a large set of similar strings. In order to improve this efficiency, we can reorder the characters in strings, so that most of the common substrings occur in the beginning or end of the string. This way, fewer nodes are needed and less memory is used. This can be implemented by a mapping between the external ordering and the actual ordering of the elements. Although it was proved that finding an optimal ordering for variables in a BDD is an NP-complete problem [5], heuristics may be used for finding a good ordering, as implemented in [1].

4.6.1 Basic Operations

LDDs support various operations which allow for efficient manipulation of sets of states and transitions. In order to define the operations on LDDs, we relate LDDs with ordered BDDs. An LDD describes a set of vectors using a graph structure which is similar to a BDD. We can make the similarities more explicit by viewing each node in an LDD as a predicate, making it possible to intuitively define LDD operations using BDD operations.

Assume an LDD describing a set of vectors $\{\langle a,b,c\rangle, \langle a,b,d\rangle, \langle b,b,c\rangle\}$. This yields the BDD from Figure 3.

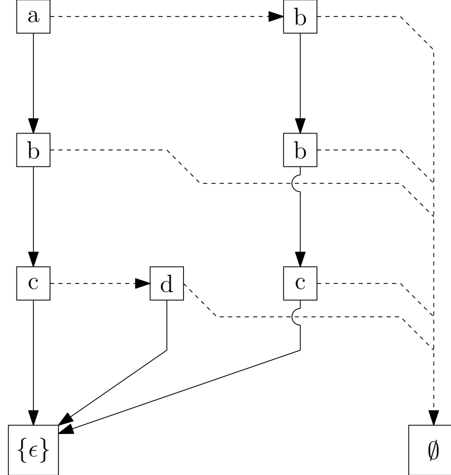


Figure 3: An LDD describing the set $\{\langle a,b,c\rangle, \langle a,b,d\rangle, \langle b,b,c\rangle\}$

We can describe the same set symbolically as follows:

$$\{v \mid (|v|=3) \wedge ((v_1=a) \wedge (v_2=b) \wedge ((v_3=c) \vee (v_3=d))) \vee ((v_1=b) \wedge (v_2=b) \wedge (v_3=c))\}$$

Ignoring the size constraint, the BDD corresponding to the guard expression is shown in Figure 4. As ordering, we use $v_1=a \lessdot v_1=b \lessdot v_2=b \lessdot v_3=c \lessdot v_{3=d}$ (ordering by vector element and then by value).

It can be seen that Figures 3 and 4 are similar. The horizontal arrows correspond to $false$ and the vertical arrows correspond to $true$. Symbols $\emptyset$ and $\{\epsilon\}$ correspond to 0 and 1. The labels l of the nodes on level n correspond to the expression $v_n = l$.

We can interpret any LDD as a BDD containing only predicates of the form $v_n = a$ where v_n is the nth element of the stored vectors and a is an element of the data sort stored in the vectors. The BDD should be ordered by the following $\lessdot$ function:

$$((v_n=a) \lessdot (v_m=b)) \Leftrightarrow (n < m \vee (n=m \wedge a \sqsubset b))$$

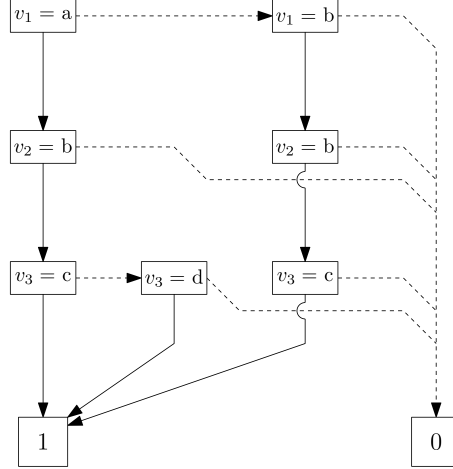


Figure 4: A BDD describing the set $\{\langle a,b,c\rangle, \langle a,b,d\rangle, \langle b,b,c\rangle\}$

Here, $\sqsubset$ is some total ordering on the elements of the alphabet.

Note that not all OBDDs following this description correspond directly to an LDD. For example, it is possible for BDDs to represent the set in which $v_1 = a \wedge v_1 = b$ holds, whereas in LDDs this can only be represented by $\emptyset$. It is possible to efficiently remove cases such as this. Because of the ordering function, we can see that this situation only occurs if the $true$ branch of a BDD node $v_n = a$ for some n and a has an edge to a BDD node $v_n = b$ for some $b \neq a$. Therefore, as a way to resolve this issue, we can replace such a node $v_n = b$ by its $false$ branch, removing its $true$ branch.

Furthermore, "gaps" in an LDD are not allowed. A node describing v_n may not have an edge to a node describing v_{n+2}, whereas in BDDs a node with the predicate $v_n = a$ may be connected directly to a node with the predicate $v_{n+2} = b$. This corresponds to the *quasi-reduced* property of BDDs ([23, 10]). Such a gap describes a situation in which any predicate of the form $v_{n+1} = c$ evaluates to $true$. Because the set described by the BDD is no longer finite (since v_{n+1} may have any value, and the universe is not known and finite), there exists no finite LDD which describes this set. In order to represent these gaps, we use the $*$ node from the definition of extended LDDs. If a node $v_n = a$ has an edge to $v_m = b$ (with $n < m$), this can be represented in an LDD by an edge from a to a string of $m - n - 1$ $*$ nodes, with the final one having an edge to b.

We only allow negated predicates ($v_1 \neq a$) if the universe for v_1 is known and finite. This allows us to perform set difference ($S - S' = S \cap \overline{S'}$) if the set from which is subtracted is finite.

The translation from an OBDD without conjunctions between nodes at the same level, and without negated predicates to an LDD (extended with $*$) follows the mapping from LDD to BDD described above.

By characterizing an LDD as a BDD, it is possible to use BDD operations in order to expand or refine the expression, implementing various set operations:

- $S \cup T$

  Since $\{s \mid \phi(s)\} \cup \{t \mid \psi(t)\} = \{u \mid \phi(u) \vee \psi(u)\}$, set union can be implemented by taking the disjunction of the two BDDs representing the sets.

- $S \sqcap T$

  We define an operation which is similar to set intersection, but ignores size constraints. This

is a type of prefix intersection.

$$(s \stackrel{prefix}{=} t) \Leftrightarrow \forall n : \mathbb{N}.1 \leqslant n \leqslant min(|s|, |t|) \wedge s_n = t_n$$
$$(s \in S \sqcap T) \Leftrightarrow (((s \in S) \wedge \exists t \in T.s \stackrel{prefix}{=} t) \vee ((s \in T) \wedge \exists t \in S.s \stackrel{prefix}{=} t)$$

This can be implemented on BDDs using a conjunction. Note that the conjunction operation may introduce BDDs such as $v_1 = a \wedge v_1 = b$. These occurrences may be removed from the BDD, by the procedure described above.

- $\overline{S}$

  Negation on a BDD is implemented by switching the 1 and 0 nodes. Similarly, negation on an LDD can be implemented by switching the $\{\epsilon\}$ and $\emptyset$ symbols.

  Note that the result of the negation of an LDD is no longer an LDD. Because the universe is not known or finite, the resulting BDD represents an infinite set, and a wildcard node can not be used. Therefore, we allow negation only for calculating set difference, where the set from which the operation subtracts must be finite. This way, the resulting set is finite and can be translated into an LDD.

- $S - T$

  Set difference (using prefix semantics, as in $\sqcap$) can be expressed using only (prefix) intersection and negation $S - T = S \sqcap \overline{T}$, and can thus be implemented on LDDs.

- $filter(S, n \leftarrow a)$

  We want to refine a set S such that only the elements (note that these are vectors) from S in which the nth element has value a: $\{v \mid v \in S \wedge v_n = a\}$. This can be achieved by putting the predicate $v_n = a$ in conjunction with the original BDD.

  We denote the operation in which an LDD S is refined with the predicate $v_n = a$ as $filter(S, n \leftarrow a)$.

- $project(S, I)$

  Projections can be done by using existential abstraction on the BDD. Using this, it is possible to describe the set $\{\langle v_2, ..., v_n\rangle \mid \exists v_1 : \langle v_1, v_2, ..., v_n\rangle \in S\}$ for some set S.

  We denote the projection of a set S on a set of indices I as $project(S, I)$.

- $swap(S, V)$

  It is possible to change the order of vector elements in an LDD. This allows for transforming a set $\{\langle v_1, v_2, v_3\rangle \mid (v_1 = a \vee v_1 = b) \wedge v_2 = c\}$ into $\{\langle v_1, v_3, v_2\rangle \mid (v_1 = a \vee v_1 = b) \wedge v_3 = c\}$. In order to do this in a BDD, which has no real concept of "levels", this requires substituting all occurrences of v_2 with v_3 and vice versa. Note that this requires an additional reordering for all substituted occurrences, in order to satisfy the ordering of the BDD.

  We denote the swapping of the elements with indices n and m in set X as $swap(X, \langle n, m\rangle)$.

- $enum(S)$

  Lastly, it is possible to enumerate all elements of a finite set. This corresponds closely to the semantics as presented in Definition 4.8. Note that when following the semantics, use of the $*$ extension as presented above will result in an infinite set, which can not be enumerated.

  By convention, we ignore $*$ values at the end of vectors. Enumerating the set $\{\langle v_1, v_2\rangle \mid v_1 = a\}$ therefore yields $\{\langle a\rangle\}$. The reason for doing this lies in the graph functions presented below. A transition function may generate a set of vectors with various lengths, but only the explicit parts of these vectors represent the actual states.

By combining these operations, it is possible to describe the functions that allow for symbolic exploration.

4.6.2 States and Transitions

We describe how states and transitions are stored in LDDs, and then define the operations which make symbolic state-space traversal possible.

We represent a state $\left\langle \mathcal{X}, \vec{d}, \mathcal{E} \right\rangle$ by the vector $\langle \mathcal{X} \rangle + \vec{d}$. A set of these vectors can be stored in an LDD. States representing $true$ and $false$ are denoted as $\langle \top \rangle$ and $\langle \bot \rangle$. The rank and decoration mappings are not stored in the state vectors, as in the given exploration method, they depend only on the first element of this vector.

The transitions are stored using an LDD as well. Instead of storing the complete set of transitions, we use local transitions as described in Section 4.4.2. Assume a PBES $\mathcal{E}$ and transition relation $\left\langle X, \vec{d}, \mathcal{E} \right\rangle \to \left\langle X', \vec{d'}, \mathcal{E} \right\rangle$. We obtain a vector $\langle v_1, v_2, ..., v_N \rangle$ (where $N = 2 \cdot \max(|\vec{d}|, |\vec{d'}|) + 2$) describing the transition relation as follows:

$$\begin{aligned} v_1 =& X \\ v_2 =& X' \\ v_n + 3 =& \begin{cases} d_{(n+1)/2} & \text{if } n \text{ is odd, } (n+1)/2 \leqslant |\vec{d}| \\ * & \text{if } n \text{ is odd, } (n+1)/2 > |\vec{d}| \\ d'_{n/2} & \text{if } n \text{ is even, } n/2 \leqslant |\vec{d'}| \\ * & \text{if } n \text{ is even, } n/2 > |\vec{d'}| \end{cases} \end{aligned}$$

For example, a transition $\langle X, \langle 0, 1 \rangle, \mathcal{E} \rangle \to \langle X', \langle 1, 1 \rangle, \mathcal{E} \rangle$ is represented by the vector $\langle X, X', 0, 1, 1, 1 \rangle$. The main reason for alternating the elements of the source and target state is that these are often related. Grouping them together possibly improves efficiency of the containing LDD.

The $*$ symbols in these transitions denote $*$ nodes in the LDD. In order to store local transitions, we also make use of this node to represent the $*$ values in the local transition. By doing this, we simplify the graph operations.

Using the defined basic LDD operations, we can define the operations that work on sets of states and transitions.

Definition 4.9 (*source* **and** *target*) *We define functions source and target for getting the source and target states from a set of transitions.*

$$\begin{aligned} source(T) =& project(T, \{2n+1 \mid n \in \mathbb{N} \wedge 2n+1 \leqslant max(|T|)\}) \\ target(T) =& project(T, \{2n+2 \mid n \in \mathbb{N} \wedge 2n+2 \leqslant max(|T|)\}) \end{aligned}$$

Note that the offset by 1 in the index is there because we only use positive natural numbers as indices. These functions are simply projections to even or odd states. It is not necessary to trim the resulting vectors to the correct length, because they are padded with $*$ nodes, which cause only a single node of overhead each, since they are placed at the end of the structure. These extra nodes do not interfere with any of the BDD operations (since they do not exist in the corresponding BDD) and thus they do not interfere with the LDD operations, which are based entirely on BDD operations.

Definition 4.10 (*explode*) *Whereas source and target remove the target or source states from a transition to obtain a set of single (partial) states, $explode_{source}$ and $explode_{target}$ introduce source or target states, transforming a state into set of transitions.*

$$\begin{aligned} explode_{source}(T) =& \{v \mid t \in T \wedge n \in \mathbb{N} \wedge n < |T| \wedge v_{2n+1} = t_{n+1} \wedge v_{2n+2} = *\} \\ explode_{target}(T) =& \{v \mid t \in T \wedge n \in \mathbb{N} \wedge n < |T| \wedge v_{2n+1} = * \wedge v_{2n+2} = t_{n+1}\} \end{aligned}$$

The *explode* functions can be implemented using *swap* operations.

Example 19 (*explode*) *Assume a state vector* $s = \langle X, 1, 2, 3 \rangle$*. Then the following holds:*

$$explode_{source}(\{s\}) = \{\langle X, *, 1, *, 2, *, 3, * \rangle\}$$

The result is a transition vector representing any transition from $X(1, 2, 3)$.

Furthermore, the following holds:

$$explode_{target}(\{s\}) = \{\langle *, X, *, 1, *, 2, *, 3 \rangle\}$$

The result of this operation is a transition vector representing any transition to $X(1, 2, 3)$.

Definition 4.11 (*match*) *In order to find out which local transitions need to be generated, we need a function match to match local transitions to the global set of source states. This function is defined as follows:*

$$match(S, S_{local}) = S \sqcap S_{local}$$

Because the $*$ *elements of the local transitions are translated into* $*$ *nodes in the LDD describing these transitions, which allows any value to be filled in for the* $*$*, matching the source or target of a local transition corresponds to taking the (prefix) intersection of the two sets.*

Example 20 (*match*) *Assume a local transition* $\langle X, \langle 1, * \rangle, \mathcal{E} \rangle \to \langle X, \langle 2, * \rangle, \mathcal{E} \rangle$*, represented by a transition vector* $t = \langle X, X, 1, 2, *, * \rangle$*. Then* $source(\{t\}) = \{\langle X, 1, * \rangle\}$*. Assume some set of states* $S = \{\langle X, 1, 2 \rangle, \langle X, 1, 3 \rangle, \langle X, 2, 1 \rangle, \langle Y, 1 \rangle\}$*. We use match to find the states in this set which are described by the source of t, by calculating* $match(S, \{\langle X, 1, * \rangle\})$*. The result of this calculation is the set of all states in S of which the first element is X and the second is* 1*. Thus* $match(S, \{\langle X, 1, * \rangle\}) = \{\langle X, 1, 2 \rangle, \langle X, 1, 3 \rangle\}$.

Definition 4.12 (*next* **and** *prev*) *We define the function next which calculates the set of next-states from a set of states, following a set of local transitions. We also define the prev function which follows the transitions backwards and obtains a set of previous-states.*

We define next and prev as follows:

$$next(S, R, I_{wi}) = target(swap(match(explode_{source}(S), R), [\langle 2i+2, 2i+1 \rangle \,|\, i \in I_{wi}]))$$
$$prev(S, R, I_{ri}) = source(swap(match(explode_{target}(S), R), [\langle 2i+2, 2i+1 \rangle \,|\, i \in I_{ri}]))$$

Here, S is the set of source state vectors, R is the set of local transitions to follow and I_{wi} *is the set of write* independent *indices, which is the set of all indices i of the target vectors for which* $i \notin changed(\mathcal{X}, n)$*. Similarly,* I_{ri} *is the set of read independent indices of the source vectors* ($i \notin used(\mathcal{X}, n)$)*.*

Function *prev* is used in the recursive algorithm for solving the resulting BES. This is described in Section 5.3.1.

Note that the set of states returned by the *prev* function is an overapproximation of the actual set of previous states. Assume a transition $\langle X, \langle a, * \rangle, \mathcal{E} \rangle \to \langle X, \langle b, * \rangle, \mathcal{E} \rangle$ and a transition $\langle X, \langle c, * \rangle, \mathcal{E} \rangle \to \langle X, \langle b, * \rangle, \mathcal{E} \rangle$. Assume initial state $\langle X, \langle a, d \rangle, \mathcal{E} \rangle$. Computing the set of next-states yields $\{\langle X, \langle b, d \rangle, \mathcal{E} \rangle\}$. Calling *prev* on this single state $\langle b, d \rangle$ will result in a set of states $\{\langle X, \langle a, d \rangle, \mathcal{E} \rangle, \langle X, \langle c, d \rangle, \mathcal{E} \rangle\}$, while the state $\langle X, \langle c, d \rangle, \mathcal{E} \rangle$ is not present in the graph. In order to solve this, the resulting set of *prev* needs to be intersected with the set of states in the graph.

Example 21 (*next*) *Assume transition vector* $t = \langle X, X, 1, 2, *, * \rangle$ *from Example 20. We can calculate the set of next-states of a state represented by the vector* $s = \langle X, 1, 2 \rangle$ *following transition*

t by computing $next(\{s\},\{t\},\{2\})$. Here, $I_{wi} = \{2\}$, because X is not write dependent on the second parameter.

First, we calculate $S' = explode_{source}(\{s\}) = \{\langle X,*,1,*,2,*\rangle\}$. We then match it to the transition using $T' = match(S',\{t\}) = \{X,X,1,2,2,*\}$. Then the source and target for the second parameter are swapped using $T'' = swap(T',[\langle 6,5\rangle]) = \{X,X,1,2,*,2\}$. Taking the target of this, we find $target(T'') = \{\langle X,2,2\rangle\}$. This is indeed the result of applying the transition $X(1,*)\rightarrow X(2,*)$ to state $X(1,2)$.

4.7 Algorithm

Using the defined operations, it is possible to write an algorithm which calculates for any set of states, the set of next-states, thereby performing a single exploration step. This algorithm, named NEXTSTATESYM, is given in Algorithm 2.

Algorithm 2 NEXTSTATESYM

```
1:  function NEXTSTATESYM(S_0, C)
2:      S := S_0 − {⟨⊤⟩, ⟨⊥⟩}                                 ▷ Ignore the two special nodes
3:      T := ∅                                                ▷ Next states
4:      for all recursion variables and clusters 𝒳 do
5:          S_𝒳 := filter(S, 1 ← 𝒳)
6:          for all transition groups n for 𝒳 do
7:              R := C[𝒳, n]
8:              S' := S_𝒳 − match(S_𝒳, source(R))   ▷ Read transitions from the cache if possible
9:              while S' ≠ ∅ do                       ▷ Find all unseen local transitions
10:                 Pick some s from S'
11:                 t := nextState_{ℰ𝒳,n}(s)
12:                 for all i ∈ used(𝒳, n) do         ▷ Generate the local transition
13:                     s_{i+1} := *
14:                 end for
15:                 for all i ∈ changed(𝒳, n) do
16:                     t_{i+1} := *
17:                 end for
18:                 u := ⟨⟩
19:                 for n ∈ ℕ, n < max(|s|, |t|) do
20:                     u_{2n+1} := { s_n   if n < |s|
                                    { *     otherwise
21:                     u_{2n+2} := { t_n   if n < |t|
                                    { *     otherwise
22:                 end for
23:                 S' := S' − match(S', {s})         ▷ Remove all other matching states from S'
24:                 R := R ∪ {u}
25:             end while
26:             C[𝒳, n] := R
27:             I_wi := {i ∈ ℕ | i < |R|/2 ∧ i ∉ changed(𝒳, n)}
28:             T := T ∪ next(S_𝒳, R, I_wi)
29:         end for
30:     end for
31:     return ⟨T, C⟩
32: end function
```

This algorithm performs a next-state calculation on a potentially large set of states S_0 (stored in an LDD) simultaneously after generating the local transition functions. It uses a cache $\mathcal{C}$,

containing for each transition group $\mathcal{X}, n$ (notation: $\mathcal{C}[\mathcal{X}, n]$), the transition relation as explored so far. This way, the results of previous calculations can be reused in a next step. The new set of local transitions is returned, and can be used as a cache in the subsequent step. Usage of this cache is optional.

The algorithm evaluates all transition groups separately. For each of these groups, the local transition functions are generated. This is done by generating the global transition function from the set of states S', and replacing the values by $*$, following the description in Section 4.4.2. All states matching this local transition are removed from S'. This way, a complete set of local transitions for the current states is formed. These local transitions are applied to the set of current states, finding the set of next states for the transition group. After following this process for all transition groups of all recursion variables and clusters, the complete set of successors of the states in S_0 is calculated.

Example 22 (nextStateSym) *Assume the clustered PBES $\mathcal{E}$ from Example 11:*

$$\begin{aligned} \nu X(b:\mathbb{B}, c:\mathbb{B}) =& \neg(b \wedge c) \Rightarrow (X(b \vee c, \neg c) \vee \exists d \in \mathbb{B}.true \wedge c_1(b, c, d)) \\ c_1(b:\mathbb{B}, c:\mathbb{B}, d:\mathbb{B}) =& X(b \wedge c, d) \wedge X(b \wedge d, c) \end{aligned}$$

In order to calculate the next-states from state $\langle X, \langle false, false\rangle, \mathcal{E}\rangle$, we use the procedure NEXTSTATESYM$(\{\langle X, false, false\rangle\}, \emptyset)$*. At the start of the algorithm, $S = \{\langle X, false, false\rangle\}$ and $T = \emptyset$.*

First, the transitions for X are evaluated. Variable S_X is assigned the set $\{\langle X, false, false\rangle\}$. Transition group 1 for this recursion variable is then used. Since $\mathcal{C}[X, 1] = \emptyset$, $S' = S_X$ and the algorithm will calculate $nextState_{\mathcal{E}X,1}(\langle X, \langle false, false\rangle, \mathcal{E}\rangle)$. This results in the set $\{\langle X, false, true\rangle\}$. This global transition is then transformed into a local transition, resulting in the equivalent transition $\{\langle X, false, false\rangle \rightarrow \langle X, false, true\rangle\}$, as there is a read and write dependency on both parameters. Then S' is empty, so all local relations have been found. The set of target states is calculated by using next, resulting in $\langle X, false, true\rangle$.

The next transition group is 2 of recursion variable X. Again, $nextState_{\mathcal{E}X,2}(\langle X, false, false\rangle)$ is calculated, resulting in $\{\langle c_1, false, false, false\rangle, \langle c_1, false, false, true\rangle\}$. Calculating the local transition yields $\{\langle X, *, *\rangle \rightarrow \langle c_1, *, *, false\rangle, \langle X, *, *\rangle \rightarrow \langle c_1, *, *, true\rangle\}$. Applying this transition results in the following set:

$$T = \{\langle X, false, true\rangle, \langle c_1, false, false, false\rangle, \langle c_1, false, false, true\rangle\}$$

Since S does not contain any c_1 states, the last two transition groups have no effect on the set of next-states. Therefore, the operation results in the following tuple:

$$\begin{aligned} &\langle\{\langle X, false, true\rangle, \langle c_1, false, false, false\rangle, \langle c_1, false, false, true\rangle\}, \\ &\quad \{X, 1 \rightarrow \{\langle X, *, *\rangle \rightarrow \langle c_1, *, *, false\rangle, \\ &\qquad\qquad \langle X, *, *\rangle \rightarrow \langle c_1, *, *, true\rangle\}, \\ &\quad X, 2 \rightarrow \{\langle X, false, false\rangle \rightarrow \langle X, false, true\rangle\} \\ &\quad c_1, 1 \rightarrow \emptyset \\ &\quad c_1, 2 \rightarrow \emptyset\}\rangle \end{aligned}$$

The first element of this tuple is a set containing all explored states, and the second element contains the found local transitions per transition group. The union of the sets of local transitions per group is the complete set of found local transitions.

Using the symbolic next-state procedure, it is possible to perform a complete exploration of the state-space, as shown in Algorithm 3.

This algorithm repeatedly calls NEXTSTATESYM until the (least) fixed-point is reached. Optionally, it caches all transitions between steps. The returned cache from the NEXTSTATESYM call

Algorithm 3 EXPLOREPBESSYM

```
1: function explorePBESSym(s0, useCache)
2:     S := {s0}                                        ▷ Encountered states
3:     T := ∅                                           ▷ Explored states
4:     C[X, n] := ∅ for all X, n                        ▷ Transition cache
5:     while S ≠ T do
6:         T := S
7:         if useCache then
8:             ⟨S, C⟩ := nextStateSym(S, C)
9:         else
10:            ⟨S, C'⟩ := nextStateSym(S, ∅)
11:            for all X, n do
12:                C[X, n] := C[X, n] ∪ C'[X, n]
13:            end for
14:        end if
15:    end while
16:    d = λs.d(s0)                                     ▷ d(⟨X, p, E⟩) = d(X)
17:    r = λs.rank_E(s0)                                ▷ r(⟨X, p, E⟩) = rank_E(X)
18:    return ⟨S, s0, C, d, r⟩
19: end function
```

is used to define the $\rightarrow$ function. The decoration and rank mappings depend only on the first element of a state node (the recursion variable or cluster name), and are defined as such. A vector $\langle S, s_0, \rightarrow_p, d, r\rangle$ is returned in which S is an LDDs as described in Section 4.6.2, s_0 is the initial state, d is a mapping from states to decorations and r is a mapping from ranked states to their ranks. The returned value $\rightarrow_p$ is the partitioned set of explored transitions. Each partition of $\rightarrow_p$ is an LDD containing a set of local transitions, as described in Section 4.6.2. The reason for returning a partitioned transition relation is that the *next* function on local transitions will only work on transitions from the same partition, because it depends on the value of *changed*.

There are several alternatives to this exploration method. While NEXTSTATESYM only calculates the set of next-states for a set of states, a complete exploration requires the set of transitive next-states. This can be calculated by repeatedly using NEXTSTATESYM, but it is possible to define an exploration procedure which may explore more than a single level of the graph at a time. One strategy is saturation ([10]), in which the transitions corresponding to a transition group are not applied once but repeatedly, until the set of next-states does not change by applying the relation. Another method is chaining, in which the resulting states for each transition group are also used in the exploration of the next group. This strategy can be implemented in Algorithm 2 by changing the assignment of S' to $S' := (S_{\mathcal{X}} \cup filter(T, 1 \leftarrow \mathcal{X})) - match(S_{\mathcal{X}}, source(R))$. Using saturation or chaining, the number of exploration steps may be reduced.

4.7.1 Usage

The instantiation algorithm from Section 4.7 results in a 5-tuple $\langle S, s_0, \rightarrow, \mathcal{C}, r\rangle$, where S is a symbolic data structure containing the set of all state vectors and $\mathcal{C}$ is a mapping from transition group $\mathcal{X}, n$ to a symbolic data structure containing all local transitions corresponding to this group.

The functions defined in Sections 4.6.1 and 4.6.2 can be used on these structures. Because S is fully explored, it is not necessary to use NEXTSTATESYM to find the set of next-states.

Definition 4.13 ($next_{\mathcal{C}}$ and $prev_{\mathcal{C}}$) *We define functions $next_{\mathcal{C}}$ and $prev_{\mathcal{C}}$ which calculate the*

set of next-states over the transitions stored in $\mathcal{C}$ as follows:

$$next_{\mathcal{C}}(S') = \bigcup_{\mathcal{X},n \in dom(\mathcal{C})} next(S', \mathcal{C}[\mathcal{X}, n], \{i \in \mathbb{N} \mid i < max\{|u| \mid u \in S'\} \wedge i \notin changed(\mathcal{X}, n)\})$$
$$prev_{\mathcal{C}}(S') = \bigcup_{\mathcal{X},n \in dom(\mathcal{C})} prev(S', \mathcal{C}[\mathcal{X}, n], \{i \in \mathbb{N} \mid i < max\{|u| \mid u \in S'\} \wedge i \notin used(\mathcal{X}, n)\})$$

It is important to note that following the local transitions backwards (as done in $prev_{\mathcal{C}}$) does not necessarily result in states that are in S, as shown in Section 4.6.2. In order to find the previous-states in S, intersecting the two sets suffices.

Similarly to NEXTSTATESYM, chaining and saturation techniques may be used to improve performance of $next_{\mathcal{C}}$.

Example 23 (*next* and *prev*) *Assume a set of local states S and a local transition mapping $\mathcal{C}$ as follows:*

$$S = \{\langle X, 1, 2\rangle, \langle X, 1, 3\rangle, \langle Y, 2, 2, 3\rangle, \langle Y, 2, 3, 4\rangle, \langle T\rangle, \langle F\rangle\}$$
$$\mathcal{C} = \left\{ \begin{array}{l} X, 1 \rightarrow \left\{ \begin{array}{l} \langle X, Y, 1, 2, *, 2, *, 3\rangle \\ \langle X, Y, 1, 2, *, 3, *, 4\rangle \end{array} \right\} \\ Y, 1 \rightarrow \left\{ \langle Y, T, *, *, *, *, *, *\rangle \right\} \end{array} \right\}$$

Node that S is fully explored over $\mathcal{C}$.

Assume a set of states $S' \subseteq S$ as follows:

$$S' = \{\langle X, 1, 2\rangle, \langle Y, 2, 3, 4\rangle, \langle T\rangle, \langle F\rangle\}$$

We compute the set of next-states $next_{\mathcal{C}}(S')$ as follows:

$$\begin{aligned} next_{\mathcal{C}}(S') =& next(S', \{\langle X, Y, 1, 2, *, 2, *, 3\rangle, \langle X, Y, 1, 2, *, 3, *, 4\rangle\}, \emptyset) \\ & \cup next(S', \{\langle Y, T, *, *, *, *, *, *\rangle\}, \emptyset) \\ =& \{\langle Y, 2, 2, 3\rangle, \langle Y, 2, 3, 4\rangle\} \cup \emptyset \\ =& \{\langle Y, 2, 2, 3\rangle, \langle Y, 2, 3, 4\rangle\} \end{aligned}$$

Similarly, the set of previous-states of this resulting set is computed as follows:

$$\begin{aligned} S'' =& \{\langle Y, 2, 2, 3\rangle, \langle Y, 2, 3, 4\rangle\} \\ prev_{\mathcal{C}}(S'') =& prev(S'', \{\langle X, Y, 1, 2, *, 2, *, 3\rangle, \langle X, Y, 1, 2, *, 3, *, 4\rangle\}, \{2\}) \\ & \cup prev(S'', \{\langle Y, T, *, *, *, *, *, *\rangle\}, \emptyset) \\ =& \{\langle X, 1, 2, *\rangle, \langle X, 1, 3, *\rangle\} \cup \emptyset \\ =& \{\langle X, 1, 2, *\rangle, \langle X, 1, 3, *\rangle\} \end{aligned}$$

Note that 3 is not included in the I_{ri} parameter to prev for group $X, 1$, because the source states of the transition have only two parameters.

Because all resulting states are included in S, it is not necessary to intersect the result with S.

4.8 BESsyness

The described steps for obtaining a structure graph from a PBES are defined such that any resulting structure graph is trivially BESsy (see Definition 2.6). BESsyness is a highly desired property, because a BESsy structure graph corresponds to a valid BES. We give a short explanation why the resulting structure graphs have the BESsyness property.

The first property which must hold on the structure graph, states that all nodes s for which $d(s) \in \{\top, \bot\}$ may not have a successor with respect to $\rightarrow$. This property is trivially validated by definition of the $true$ and $false$ nodes, as given in Definition 4.1. This definition explicitly states that $nextState_{\mathcal{E}}$ for these nodes is equal to $\emptyset$, and thus they have no successors in the resulting structure graph.

The second property states that for all nodes s, $d(s) \in \{\wedge, \vee\}$ or $r(s)$ is defined, if and only if s has a successor with respect to $\rightarrow$. This property is also trivially validated by Definition 4.1, because it follows directly from the definition of $nextState_{\mathcal{E}}$ that all states except $true$ and $false$ (which do not have a rank) have at least one successor with respect to $\rightarrow$. Furthermore, as explained in Section 4.2 and enforced in Algorithms 1 and 3, $d(s)$ is always assigned either $\wedge$ or $\vee$, except for the $true$ and $false$ nodes. As a result, this property is true for both the $true$ and $false$ nodes and the other nodes.

The third property of BESsy structure graphs states that $d(s)$ must be equal to $\wedge$ or $\vee$ for all nodes s with multiple successors with respect to $\rightarrow$. As explained above, this is trivially valid on the generated structure graphs, because $d(s) \in \{\wedge, \vee\}$ holds for all nodes with at least one successor.

The final property states that all cycles with respect to $\rightarrow$ contain at least one ranked node. Ranked nodes, as explained in Section 4.2, correspond to nodes that were instantiated from equations (as opposed to clusters).

Assume that a cycle $\left\langle c_a, \vec{d_a}, \mathcal{E} \right\rangle, \left\langle c_b, \vec{d_b}, \mathcal{E} \right\rangle, ..., \left\langle c_z, \vec{d_z}, \mathcal{E} \right\rangle, \left\langle c_a, \vec{d_a}, \mathcal{E} \right\rangle$ of unranked nodes exists. Then this cycle must contain at least one transition from $\left\langle c_i, \vec{d_i}, \mathcal{E} \right\rangle \rightarrow \left\langle c_j, \vec{d_j}, \mathcal{E} \right\rangle$ such that $i \geqslant j$. Thus, the clustered PBES from which the structure graph is generated must contain a cluster c_i with a reference to cluster c_j such that $i \geqslant j$. Following the *Cluster* procedure, defined in Definition 3.1, we find that clusters only refer to newly generated "fresh" clusters, and therefore a cluster c_i can only refer to a cluster c_j if $i < j$. From this contradiction it follows that the structure graph does not contain cycles of unranked nodes.

4.9 Optimizations

Operations exist for optimizing a structure graph. These operations are based on BES transformations for simplifying a system. Some of these optimizations, such as true/false-elimination, can be applied efficiently on a symbolic structure graph.

4.9.1 True/false-elimination

True/false-elimination is the operation which performs the following boolean transformations:

$$
\begin{aligned}
b \wedge true &\rightarrow b \\
b \wedge false &\rightarrow false \\
b \vee true &\rightarrow true \\
b \vee false &\rightarrow b
\end{aligned}
$$

On a structure graph, the following transformations achieve this:

1. All transitions to a $true$-node from a conjunctive node with at least one other transition are removed.
2. All transitions to a $false$-node from a disjunctive node with at least one other transition are removed.
3. All unranked conjunctive nodes with a transition to a $false$-node are replaced by $false$.

4. All unranked disjunctive nodes with a transition to a *true*-node are replaced by *true*.
5. For ranked conjunctive nodes with a transition to a *false*-node all other outgoing transitions are removed.
6. For ranked disjunctive nodes with a transition to a *true*-node all other outgoing transitions are removed.
7. All unranked nodes with a single transition to the *true*-node are replaced by this node.
8. All unranked nodes with a single transition to the *false*-node are replaced by this node.

These transformations are quite complex, compared to the regular set of boolean optimizations, due to the added difficulty of preserving BESsyness. This preservation is required in order to be able to transform the result to a valid BES.

All of the transformations require removal of specific transitions from the graph. Because in the described symbolic instantiation approach, only local transitions are stored, it is not easy to remove specific transitions from the graph. Furthermore, the removed transitions form an explicit set of global transitions. These global transitions had been avoided during the instantiation, in order to improve efficiency. Experiments have to be done in order to determine whether applying these optimizations is useful in practice.

We describe two approaches for removing transitions and we give algorithms to calculate the exact transitions to be removed.

In order to be able to remove transitions, it is possible to transform the set of local transitions into global transitions. This simplifies the method, but for large systems this may not be feasible. Getting a set of global transitions $\rightarrow_G$ from a set of local transitions $\rightarrow$ and states S is done by calculating $\rightarrow_G := \rightarrow \sqcap explode_{source}(S) \sqcap explode_{target}(S)$. Repeating this for the transitions in all transition groups and taking the union of the results, the complete set of global transitions is found. From a set of global transitions, the set difference operator can be used safely for removing transitions.

The second approach to removing transitions is to lift the restrictions that prevent negative predicates ($v_1 \neq a$) to exist in the BDD corresponding to the symbolic structure. Following this restriction, the set of transitions can no longer necessarily be transformed into an LDD, because it may contain negations. The *next* and *prev* operations still work as expected. Note that when removing transitions from the graph, they have to be removed for all transition groups. However, the resulting symbolic structure may be unexpected. A rather complex structure may represent a transition which in practice never occurs. This happens when all occurring transitions corresponding to a certain transition group have been removed.

In addition to removing transitions, some operation require transitions to be added. We add these transitions to a new transition group with only explicit transitions.

When allowing transitions to be added and removed, the following procedures correspond to the transformations described above. We evaluate a symbolic structure graph $\langle S, s_0, \mathcal{C}, d, r\rangle$.

1. First, we find all conjunctive states with a transition to the *true* state.

   $$S_{prev} = filter(prev_{\mathcal{C}}(\{\langle\top\rangle\}), 1 \leftarrow \{\mathcal{X} \in \mathbf{X} \mid d(\mathcal{X}) = \wedge\})$$

   We find all states, except the *true* state, to which the states in S_{prev} have a transition as follows:

   $$S' = next_{\mathcal{C}}(S_{prev}) - \{\langle\top\rangle\}$$

   By intersecting the previous states of S' with S_{prev}, we find the set of conjunctive states which have another transition than the one to the *true* state:

   $$S'_{prev} = S_{prev} \sqcap prev_{\mathcal{C}}(S')$$

From this, we build the set of transitions to remove:

$$\rightarrow_{rem} = explode_{source}(S'_{prev}) \sqcap explode_{target}(\{\langle\top\rangle\})$$

2. Symmetric to the previous procedure.

3. First, we find all unranked conjunctive states with a transition to the $false$ state.

$$S_{prev} = filter(prev_{\mathcal{C}}(\{\langle\bot\rangle\}), 1 \leftarrow \{\mathcal{X} \in \mathbf{X} \mid d(\mathcal{X}) = \wedge \wedge \mathcal{X} \notin dom(rank_{\mathcal{E}})\})$$

These nodes should be replaced by the $false$ node. To do this, we find their predecessors:

$$S_{prev^2} = prev_{\mathcal{C}}(S_{prev})$$

We remove the transition from states in S_{prev^2} to states in S_{prev} and add a transition from the states in S_{prev^2} to the $false$ state.

$$\begin{aligned}\rightarrow_{rem} &= explode_{source}(S_{prev^2}) \sqcap explode_{target}(S_{prev})\\ \rightarrow_{add} &= explode_{target}(S_{prev^2}) \sqcap explode_{target}(\{\langle\bot\rangle\})\end{aligned}$$

4. Symmetric to the previous procedure.

5. First, we find all ranked conjunctive states with a transition to the $false$ state.

$$S_{prev} = filter(prev_{\mathcal{C}}(\{\langle\bot\rangle\}), 1 \leftarrow \{\mathcal{X} \in \mathbf{X} \mid d(\mathcal{X}) = \wedge \wedge \mathcal{X} \in dom(rank_{\mathcal{E}})\})$$

All transitions except the one to $false$ should be removed. We find these transitions as follows:

$$S' = next_{\mathcal{C}}(S_{prev}) - \langle\top\rangle$$

$$\rightarrow_{rem} = explode_{source}(S'_{prev}) \sqcap explode_{target}(S')$$

6. Symmetric to the previous procedure.

7. First, we find all unranked states with a transition to the $true$ state.

$$S_{prev} = filter(prev_{\mathcal{C}}(\{\langle true\rangle\}), 1 \leftarrow \{\mathcal{X} \in \mathbf{X} \mid \mathcal{X} \notin dom(rank_{\mathcal{E}})\})$$

We find all states, except the $true$ state, to which the states in S_{prev} have a transition as follows:

$$S' = next_{\mathcal{C}}(S_{prev}) - \{\langle\top\rangle\}$$

By subtracting the previous states of S' from S_{prev}, we find the set of states which do not have another transition than the one to the $true$ state:

$$S'_{prev} = S_{prev} - prev_{\mathcal{C}}(S')$$

These nodes should be replaced by the $true$-node. To do this, we find their predecessors:

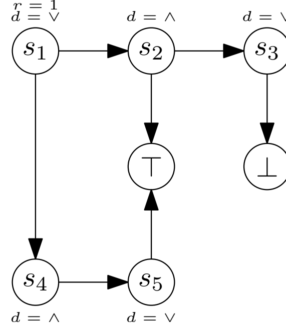
$$S_{prev^2} = prev_{\mathcal{C}}(S_{prev})$$

We remove the transition from states in S_{prev^2} to states in S_{prev} and add a transition from the states in S_{prev^2} to the $true$ state.

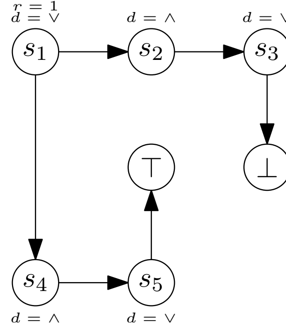
$$\begin{aligned}\rightarrow_{rem} &= explode_{source}(S_{prev^2}) \sqcap explode_{target}(S_{prev})\\ \rightarrow_{add} &= explode_{target}(S_{prev^2}) \sqcap explode_{target}(\{\langle\top\rangle\})\end{aligned}$$

8. Symmetric to the previous procedure.

Example 24 (True/false-elimination) *We evaluate the following structure graph:*



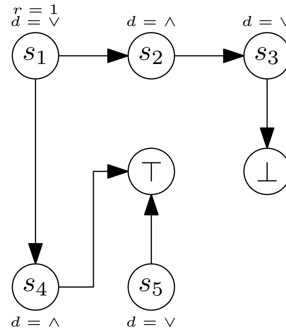
We use the described simplification steps on this graph. We first apply the first optimization step. We find $S_{prev} = \{s_2\}$*,* $S' = \{s_3\}$ *and* $S'_{prev} = \{s_2\}$*, and thus remove the edge from* s_2 *to* $\top$*. This results in the following graph:*



The second step does not modify the graph, because $S'_{prev} = \emptyset$*.*

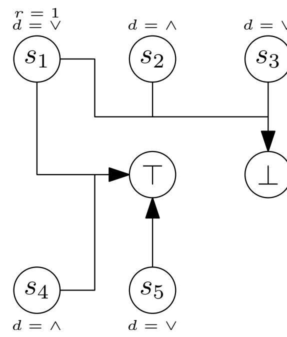
In the third step, we find that $S_{prev} = \emptyset$*, and thus the graph is not modified.*

In the fourth step, $S_{prev} = \{s_5\}$*,* $S_{prev^2} = \{s_4\}$*, and as a result, the transition from* s_4 *to* s_5 *is removed and a transition from* s_4 *to* $\top$ *is added:*

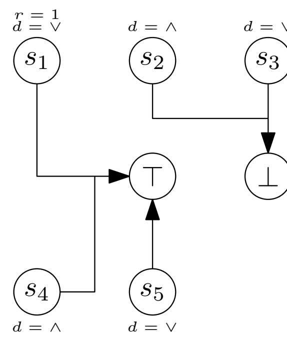


In the fifth and sixth step, $S_{prev} = \emptyset$*, so the graph remains the same.*

For the seventh step, $S_{prev} = \{s_4, s_5\}$*,* $S' = \emptyset$*,* $S'_{prev} = \{s_4, s_5\}$ *and* $S_{prev^2} = \{s_1\}$*. We therefore remove the edge from* s_1 *to* s_4 *and* s_1 *to* s_5 *(ignoring the fact that this last edge does not exist) and add an edge from* s_1 *to* $\top$*. Step eight is done in a similar way, resulting in the following structure graph:*



Executing the fifth step on this structure graph results in the following optimized graph:



It can be seen that the ranked node only retains an edge to the $\top$ *node, so the equation corresponding to this node can not be optimized any more.*

5 Solving

After instantiating a PBES to a BES, the BES can be solved using one of various algorithms from the literature. We first relate structure graphs to parity games ([14]). Following this relation, we extend the recursive algorithm for solving parity games to the realm of symbolic structure graphs, in order to apply the algorithm directly on the instantiation results. In addition, we describe how the Gauß elimination algorithm for solving BESs ([24]) extends to symbolic structure graphs.

5.1 Parity Games

Parity games are graph games which have a direct correspondence with boolean equation systems in simple recursive form. We define parity games, and relate them to structure graphs.

5.1.1 Simple Recursive Form

Simple Recursive Form (SRF) is a normal form for BESs, in which all equations must be fully conjunctive or disjunctive.

This limits the form of Φ, changing the definition of a BES (see Section 2.1) as follows:

$$\begin{aligned} BES_{SRF} ::=& (\sigma\mathcal{X} = \Phi_{SRF}) \\ & \mid (\sigma\mathcal{X} = \Phi_{SRF})\ BES \\ \Phi_{SRF} ::=& \Phi_{\wedge} \mid \Phi_{\vee} \\ \Phi_{\wedge} ::=& \mathcal{X} \mid \Phi_{\wedge} \wedge \Phi_{\wedge} \\ \Phi_{\vee} ::=& \mathcal{X} \mid \Phi_{\vee} \vee \Phi_{\vee} \end{aligned}$$

Example 25 (SRF) *The following BES is not in SRF:*

$$\begin{aligned} \nu X =& X \wedge (Y \vee Z) \\ \mu Y =& Y \\ \nu Z =& Y \end{aligned}$$

The reason for this is that the equation for X contains both a conjunction and a disjunction.

Every BES can be rewritten into a BES in SRF with an equal solution, by introducing new equations. The previous example can be rewritten to SRF as follows:

$$\begin{aligned} \nu X =& X \wedge X' \\ \nu X' =& Y \vee Z \\ \mu Y =& Y \\ \nu Z =& Y \end{aligned}$$

Note that the solution for X is unchanged.

It should also be remarked that, due to the fact that SRF does not allow $true$ and $false$ to occur in the equations, in the transformation to SRF, these are often introduced as special equations $\nu T = T$ and $\mu F = F$. When solving a boolean equation system for one of these variables, the result for F is $false$ and for T is $true$.

BESs in SRF have a one-to-one mapping with parity games ([14]). Parity games are two-player graph games where each node corresponds to a variable and the set of outgoing edges for a node corresponds to the variables occurring in the right-hand side of that formula. Additionally, all nodes have a priority ($\in \mathbb{Z}$) associated to them, which represents both the ordering of the formulas and their fixed-point symbols (ν is even, μ is odd) and they have shapes ($\Box$ or $\diamond$) which describe whether they are conjunctive or disjunctive respectively.

Definition 5.1 (Parity game) *We define a parity game* Γ *as a tuple* $\langle\langle S_\diamond, S_\Box\rangle, s_0, \rightarrow, p\rangle$ *where:*

- $S_\diamond$ *is a finite set of nodes with the shape* $\diamond$
- $S_\Box$ *is a finite set of nodes with the shape* $\Box$
- $S_\diamond \cap S_\Box = \emptyset$
- s_0 *is the initial node*
- $\rightarrow \subseteq (S_\diamond \cup S_\Box) \times (S_\diamond \cup S_\Box)$ *is the set of edges*
- $p : (S_\diamond \cup S_\Box) \rightarrow \mathbb{Z}$ *is the priority function*

The priority function corresponds with the $rank_\mathcal{E}$ function from Definition 2.4.

Figure 5 shows the parity game corresponding to the BES from Example 25.

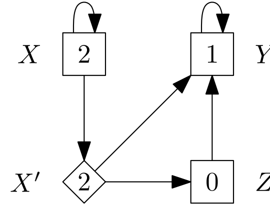


Figure 5: A parity game which corresponds to the BES from Example 25.

When playing a parity game, a token is moved over the edges of the graph by two players, one of which is named *even* and the other named *odd*. The shape of the node the token is currently on determines which of the players may make the next move ($\Box$ is *odd*, $\diamond$ is *even*). A game is won by a player P when the token is moved over an infinite path in which the highest occurring priority corresponds to the name of P.

Solving a parity game corresponds to solving the BES it represents. A parity game is solved by finding a strategy in the graph game for one of the two players such that by following it, this player inevitably wins the game.

Algorithms exist to solve parity games. Examples of these algorithms are the recursive algorithm ([32]) and small progress measures ([19]).

5.2 Transformation to Parity Games

The explored BES, which is in structure graph form, can be translated into a parity game. This translation is useful, because various algorithms for solving parity games exist. An algorithm to do this is given in [22]. This algorithm works only under the condition that the structure graph is *BESsy* (see Definition 2.6). Section 4.8 shows that this is the case for the structure graphs that are generated by the described instantiation approach.

The transformation algorithm corresponds to the transformation of a BES into Simple Recursive Form, through a process called normalization. The resulting structure graph of a BES in SRF is then easily translated into a parity game.

The transformation of a BES to SRF consists of separating conjunctive and disjunctive subexpressions, by means of introduction of additional equations for these subexpressions. In a structure graph, this corresponds to transforming unranked nodes, decorated with $\wedge$ or $\vee$ into ranked nodes. There are a few options for the ranks. A valid option would be to assign a cluster the rank of its containing equation. A simpler option is to always assign the lowest occurring rank to all unranked nodes. Note that it is not necessary for a node to already exist with this rank. Furthermore, the

used SRF form does not have a notion of *true* and *false*. We represent these as the additional equations $(\nu\top = \top)(\mu\bot = \bot)$. This translates to a structure graph as two special nodes, each with a self-loop, where the $\top$ node has an even rank and the $\bot$ node has an odd rank. Thus, we can normalize a BESsy structure graph to SRF by changing r to r' and $\rightarrow$ to $\rightarrow'$ as follows:

$$r'(\mathcal{X}) := \begin{cases} r(\mathcal{X}) & \text{if } r(\mathcal{X}) \text{ is defined} \\ 1 & \text{if } r(\mathcal{X}) \text{ is not defined and } d(\mathcal{X}) = \bot \\ 0 & \text{otherwise} \end{cases}$$
$$\rightarrow' := \rightarrow \cup \{\langle\top,\top\rangle, \langle\bot,\bot\rangle\}$$

It is shown that this approach is sound in [22].

After normalization, transformation of a structure graph $\mathcal{G} = \langle S, s_0, \rightarrow, d, r'\rangle$ into a parity game $\Gamma = \langle\langle S_\diamond, S_\Box\rangle, s'_0, \rightarrow'', p\rangle$ is done as follows:

$$\begin{aligned} S_\diamond &= \{s \in S | d(s) \in \{\vee, \bot\}\} \\ S_\Box &= \{s \in S | d(s) \in \{\wedge, \top\}\} \\ s'_0 &= s_0 \\ \rightarrow'' &= \rightarrow' \\ p &= r' \end{aligned}$$

Note that in this obtained parity game, sets $S_\diamond$ and $S_\Box$ are LDDs, and $\rightarrow''$ is a set of local transitions stored in an LDD.

By applying this transformation on the structure graph, it becomes possible to use existing algorithms for parity games to executed on the results.

We may use the information about the relation between BESsy structure graphs and parity games in order to apply algorithms that are used for solving parity games, directly to a BESsy structure graph.

5.3 Directly

We describe some solving algorithms and how they work on structure graphs. By optimizing these algorithms for use with the LDD structure, we can obtain efficient ways to solve BESs.

5.3.1 Recursive Algorithm

The recursive algorithm for solving parity games, following from [32], is an algorithm that makes use of induction on the ranks in parity games, in order to solve the games.

We make use of the relation between parity games and BESsy structure graphs to rewrite this algorithm for structure graphs.

Attractor Set Central to the recursive algorithm is the concept of an attractor set. An attractor set is a set in which a certain outcome is possible or inevitable. We extend the notion of attractor sets from parity games to structure graphs.

Definition 5.2 (Attractor set) *We define function to calculate the attractor set of a set of structure graph nodes. Assume a structure graph $\mathcal{G} = \langle S, s_0, \rightarrow, d, r\rangle$. We calculate the $\bigcirc$-attractor set (with $\bigcirc \in \{\wedge, \vee\}$) of a set of nodes S' as follows:*

$$
\begin{aligned}
Attr^n_{\bigcirc}(\mathcal{G}, S') =& S' \\
Attr^{n+1}_{\bigcirc}(\mathcal{G}, S') =& Attr^n_{\bigcirc}(\mathcal{G}, S') \\
& \cup \big\{ s \in S' \mid d(s) = \bigcirc \wedge \exists s' \in S. s \to s' \wedge s' \in Attr^n_{\bigcirc}(\mathcal{G}, S') \big\} \\
& \cup \big\{ s \in S' \mid d(s) \notin \{\bigcirc, \top, \bot\} \wedge \forall s' \in S. s \to s' \Rightarrow s' \in Attr^n_{\bigcirc}(\mathcal{G}, S') \big\} \\
Attr_{\bigcirc}(\mathcal{G}, S') =& \bigcup_{n \in \mathbb{N}} Attr^n_{\bigcirc}(\mathcal{G}, S')
\end{aligned}
$$

We make a case distinction on the decoration of a state. If $d(s)$ is undefined, it is unimportant whether it is used in the first or second set, because states with undefined decoration have exactly one next-state, and thus the $\exists$ and $\forall$ quantifiers coincide. If $d(s) \in \{\top, \bot\}$, there are no successors. The universally quantified subset would always include these nodes. This is undesired, because there is no path from these nodes to any state in S' and therefore these nodes should not be included in the attractor set for these states, unless they are included in S'.

We can use a simple algorithm to calculate the attractor set of a set of nodes using LDD operations on an explored structure graph state-space.

Algorithm 4 ATTRACTORSETSYM

```
1: function ATTRACTORSETSYM(◯, ⟨S, s_0, C, d, r⟩, S')
2:     T := S'                                    ▷ We accumulate the attractor set in T
3:     T' := ∅
4:     X := project(S, {0})
5:     X_◯ := {⟨𝒳⟩ | ⟨𝒳⟩ ∈ X ∧ d(𝒳) = ◯}
6:     X_◯̄ := {⟨𝒳⟩ | ⟨𝒳⟩ ∈ X ∧ d(𝒳) ∉ {◯, ⊤, ⊥}}
7:     while T ≠ T' do                            ▷ Calculate fixed point
8:         T' := T
9:         T_1 := filter(prev_C(T), 1 ← X) ⊓ S    ▷ Predecessors with decoration ◯
10:        T := T ∪ T_1
11:        T_p := filter(prev_C(T), 1 ← X_◯̄) ⊓ S  ▷ Predecessors with decoration ∉ {◯, ⊤, ⊥}
12:        T_2 := T_p − prev_C(next_C(prev_C(T)) − S) ▷ In T_p but no predecessors of any other node
13:        T := T ∪ T_2
14:    end while
15:    return T
16: end function
```

Algorithm Using the algorithm for attractor sets, we can build the complete recursive algorithm. This algorithm, shown in Algorithm 5, is very similar to the general recursive algorithm. The main difference is that the attractor sets are generated using LDD operations.

The result of the recursive algorithm is a tuple $\langle Z_\wedge, Z_\vee \rangle$ of sets of states. The first element of this tuple contains all states for which the semantics or interpretation is *true*. The second element is the set of states for which the semantics or interpretation evaluate to *false*. It holds that $S = Z_\wedge \cup Z_\vee$ and $Z_\wedge \cap Z_\vee = \emptyset$.

In order to solve a symbolic structure graph $\langle S, s_0, \mathcal{C}, d, r \rangle$, it is sufficient to test whether $s_0 \in recursiveSym(\langle S, s_0, \mathcal{C}, d, r \rangle)_0$.

5.3.2 Gauß Elimination

In addition to the recursive algorithm, which is generally applied to parity games, we evaluate the Gauß elimination algorithm. Gauß elimination ([24]) is an algorithm for solving BESs. It consists of four steps which must be applied repeatedly, until an answer is found:

Algorithm 5 RECURSIVESYM

```
1: function RECURSIVESYM(⟨S, s_0, C, d, r⟩)
2:     if S = ∅ then
3:         return ⟨∅, ∅⟩
4:     end if
5:     X := project(𝒳, S)
6:     m := max {r(𝒳)|⟨𝒳⟩ ∈ X}
7:     ◯, ◯̄ := { ∧, ∨   if m is even or undefined
                { ∨, ∧   otherwise
8:     X := { {X ∈ X | r(X) = m}     if m is defined
            { {c_i | c_i is a cluster}  otherwise
9:     T := filter(S, 1 ← X)
10:    A := attractorSetSym(◯, G, T)
11:    ⟨X_∧, X_∨⟩ := recursiveSym(⟨S − A, s_0, →, d, r⟩)
12:    if X_◯ = ∅ then
13:        Z_◯ := A ∪ S_◯
14:        Z_◯̄ := ∅
15:    else
16:        B := attractorSetSym(◯̄, G, S_◯̄)
17:        ⟨Y_∧, Y_∨⟩ := recursiveSym(⟨S − B, s_0, →, d, r⟩)
18:        Z_◯ := Y_◯
19:        Z_◯̄ := B ∪ Y_◯̄
20:    end if
21:    return ⟨Z_∧, Z_∨⟩
22: end function
```

1. Local resolution

   All occurrences of a recursion variable $\mathcal{X}$ with fixed-point ν are replaced by $true$ in the defining equation of $\mathcal{X}$, and all occurrences of a recursion variable $\mathcal{X}'$ with fixed-point μ are replaced by $false$ in the defining equation of $\mathcal{X}'$.

2. Boolean simplification

   Operations for simplifying boolean equations must be done. The following is a minimal set of operations:

   - $f \wedge false \rightarrow false$
   - $f \wedge true \rightarrow f$
   - $f \vee true \rightarrow true$
   - $f \vee false \rightarrow f$

3. Substitution to the left

   All occurrences of $\mathcal{X}'$ in equations to the left of the defining equation for $\mathcal{X}'$ are replaced by the right-hand side of this defining equation.

4. Global substitution of closed equations

   If the right-hand side of an equation for $\mathcal{X}$ is $true$ or $false$, all occurrences of $\mathcal{X}$ in the right-hand sides of equations may be replaced by this value.

We describe possible implementations on symbolic structure graphs for all steps.

Local resolution We start with local resolution. Local resolution on structure graphs corresponds to the detection of cycles containing exactly one ranked node. The edge in the cycle leading to the node must be replaced by an edge to *true* or $false$, depending on the rank of the node. In general, detection of variable-length cycles using the available operators is hard. We can define a process for finding these cycles by using an extension to the $next_{\mathcal{C}}$ function called $next_{Trans,\mathcal{C}}$. Whereas the $next_{\mathcal{C}}$ function takes a set of state vectors as a parameter, $next_{Trans,\mathcal{C}}$ takes a set of interlaced 3-tuples as a parameter and returns a similar set. This must be done in such a way that the first two vectors in the tuple remain unchanged, and the third elements contain the next-states of the second elements. We can use this operation to find loops in the graph.

To use these interlaced 3-tuples, we introduce functions similar to *source*, *target* and *explode*:

- *source*3 is like *source*, and returns the first element of the interlaced 3-tuples
- *target*3 is like *target*, and returns the second element of the interlaced 3-tuples
- *result*3 returns the 3rd element of interlaced 3-tuples

To mirror these operations, we introduce $explode_{source3}$, $explode_{target3}$ and $explode_{result3}$.

Algorithm 6 finds the cycles in the graph. It returns a set of transition vectors such that the target of the transition vector is the ranked node in the cycle, and the source is its predecessor in the cycle. By removing the resulting transitions and replacing them by *true* or $false$, depending on the rank of state, local resolution can be done.

Algorithm 6 CYCLESYM

```
1: function CYCLESYM(⟨S, s_0, C, d, r⟩)
2:     S_X := filter(S, 1 ← {X | X ∈ X})                          ▷ Ranked nodes
3:     T := explode_source3(S_X) ⊓ explode_result3(S_X)
4:     U := explode_source3(S_X) ⊓ explode_target3(S_X)
5:     Cycles := ∅
6:     U := next_Trans,C(U)
7:     repeat
8:         R := U ⊓ T
9:         Cycles := Cycles ∪ explode_source(target3(R)) ⊓ explode_target(result3(R))
10:        U := U − explode_result3(S_X)               ▷ Remove if the result is ranked
11:        U := swap(u, [⟨i, i+1⟩ | i mod 3 = 2])         ▷ Swap target and result
12:        U := next_Trans,C(U)
13:    until U = ∅
14:    return Cycles
15: end function
```

Boolean simplification The second step, which is boolean simplification of the equations, is described thoroughly in Section 4.9.1.

Substitution to the left The third step is the substitution of equations to the left. In a graph setting, this corresponds to substituting an edge to a ranked node by an edge to an unranked clone of this node. This operation is done by evaluating each recursion variable name $\mathcal{X}$ separately. First, it is necessary to find nodes such that they have an edge to $\mathcal{X}$, and the subformula they represent is not part of an equation with lower rank than $rank_{\mathcal{E}}(\mathcal{X})$. We can use a procedure similar to CYCLESYM to find all first ranked predecessors of the predecessors of nodes for $\mathcal{X}$. The predecessors can then be filtered to include only the ones for which the first ranked predecessors have a rank which is greater than or equal to $rank_{\mathcal{E}}(\mathcal{X})$. From those nodes, a transition to $\mathcal{X}$ must be replaced by a transition to c_* with the same parameters. In addition to this, all transitions from the original target node must be copied to the newly introduced node.

Global substitution of closed equations The fourth step is similar to steps 7 and 8 of the boolean optimizations from Section 4.9, except that it is applied to ranked nodes instead of unranked nodes.

Applying these steps until a fixed-point will result in a symbolic structure graph, such that $next_{\mathcal{C}}(\{s_0\}) = \{\langle\top\rangle\}$ if and only if the resulting value for the initial state is *true*. As described in Section 4.9, it is likely that the described operations do not perform well in practical environments. An implementation of the described algorithms will provide the required feedback for determining whether it is useful to further investigate Gauß elimination on symbolic structure graphs.

6 Conclusion

A complete approach was given for symbolic instantiation of parametrized boolean equation systems to boolean equation systems and the subsequent solving of these systems.

BES structure graphs were chosen in order to define the symbolic structure containing the result of the instantiation. Based on this choice, the notion of clusters was defined, following the structure of BES structure graphs. Functions were defined for identification of these clusters. In addition to this, a preprocessing step was defined in order to overcome problems with existing strategies for symbolic instantiation of PBESs. All transformations occurring in this step are proved to be sound.

A complete algorithm for the instantiation of the clustered PBES into a symbolic BES structure graph was given. This instantiation makes use of various techniques from the area of symbolic state-space exploration.

Two algorithms were described for solving the resulting BESs, stored in a symbolic structure. First, the recursive algorithm was extended from the realm of parity games to symbolic BES structure graphs. In addition to this, Gauß elimination was defined on structure graphs.

Following the described steps allows for the instantiation and subsequent solving of a PBES with a finite instantiation, using symbolic techniques.

7 Future Work

The objective of this work is a workable initial setup which allows for the implementation of symbolic instantiation techniques into the mCRL2 toolset[4]. The most important piece of future work is the actual implementation of the described techniques into this toolset, and performing an analysis of their practical applicability. In particular, comparing the implementation of this approach to implementations of other approaches, such as explicit exploration using mCRL2 's explicit tools pbes2bool and pbespgsolve, as well as the symbolic tool pbes2lts-sym, present in the LTSMin toolset ([4]).

Using an implementation, experiments should be done in order to determine the efficiency of various approaches. The various used structures leave room for modifications. For instance the cluster form, which describes an arbitrarily deep structure, may in practice benefit from a limited depth. The results of the instantiation when the depth is limited are structure graphs with more nodes, but dependencies may in some cases be limited by restricting depth, yielding a more compact representation of the transition relation. Furthermore, experiments can be done around the merging and reordering in transition groups.

An interesting alternative to the presented approach may be researched. Instead of rewriting the PBES into the clustered form in order to simplify the transition to structure graphs, a different approach may be investigated. It may be possible to make use of partitioned transition relations directly on a PBES in GNF (strongly guardedness is not required). Using these partitioned relations, instantiation can be done. This instantiation forms a graph in which each node represents a single equation. This can not be used directly, because structure is lost (an issue which is addressed in this work by clusters). However, because the transition groups have separate sets of transitions, it is possible to recreate the structure after the instantiation. This can be done by combining the results for the different transition groups that represent variables used in the same level of the equation into separate nodes. This alternative strategy circumvents the clustering and preprocessing, but requires some more bookkeeping, as well as manipulations on the instantiated graph, in order to form a valid structure graph.

Finally, other solving algorithms (of which an overview can be found in [21]) should be investigated, to see whether they can make efficient use of the symbolic data structure. In particular, an extension of the small progress measures ([19]) algorithm to symbolic BES structure graphs may be a promising solving strategy.

[4] `http://mcrl2.org/`

References

[1] D. Beyer, C. Lewerentz, and A. Noack. Rabbit: A tool for bdd-based verification of real-time systems. In *Computer Aided Verification*, pages 122–125. Springer, 2003.

[2] S. Blom and J. van de Pol. Symbolic reachability for process algebras with recursive data types. *Theoretical Aspects of Computing-ICTAC 2008*, pages 81–95, 2008.

[3] S. Blom, J. Van De Pol, and M. Weber. Bridging the gap between enumerative and symbolic model checkers. 2009.

[4] S. Blom, J. van de Pol, and M. Weber. Ltsmin: Distributed and symbolic reachability. In *Computer Aided Verification*, pages 354–359. Springer, 2010.

[5] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *Computers, IEEE Transactions on*, 45(9):993–1002, 1996.

[6] R. Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986.

[7] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys (CSUR)*, 24(3):293–318, 1992.

[8] J. Burch, E.M. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. *Computer Science Department*, page 435, 1991.

[9] T. Chen, B. Ploeger, J. Van De Pol, and T. Willemse. Equivalence checking for infinite systems using parameterized boolean equation systems. *CONCUR 2007–Concurrency Theory*, pages 120–135, 2007.

[10] G. Ciardo and A. Yu. Saturation-based symbolic reachability analysis using conjunctive and disjunctive partitioning. *Correct Hardware Design and Verification Methods*, pages 146–161, 2005.

[11] E. Clarke, K. McMillan, S. Campos, and V. Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–422. Springer, 1996.

[12] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking*. MIT press, 2000.

[13] M. del Mar Gallardo, C. Joubert, and P. Merino. Implementing influence analysis using parameterised boolean equation systems. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 416–424. IEEE, 2006.

[14] E.A. Emerson and C.S. Jutla. Tree automata, mu-calculus and determinacy. In *Foundations of Computer Science, 1991. Proceedings., 32nd Annual Symposium on*, pages 368–377. IEEE, 1991.

[15] J. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. *Algebraic Methodology and Software Technology*, pages 74–90, 1999.

[16] J.F. Groote and T. Willemse. Parameterised boolean equation systems. *Theoretical Computer Science*, 343(3):332–369, 2005.

[17] Y. Hwong, V. Kusters, and T. Willemse. Analysing the control software of the compact muon solenoid experiment at the large hadron collider. *Fundamentals of Software Engineering*, pages 174–189, 2012.

[18] S. Janssen. Tools for parameterized boolean equation systems, 2008.

[19] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, pages 290–301. Springer, 2000.

[20] G. Kant and J. Van de Pol. Efficient instantiation of parameterised boolean equation systems to parity games. GRAPHITE 2012, 2012.

[21] J. Keiren. An experimental study of algorithms and optimisations for parity games, with an application to boolean equation systems, 2009.

[22] J. Keiren, M. Reniers, and T. Willemse. Structural analysis of boolean equation systems. *ACM Transactions on Computational Logic (TOCL)*, 13(1):8, 2012.

[23] S. Kimura and E.M. Clarke. A parallel algorithm for constructing binary decision diagrams. In *Computer Design: VLSI in Computers and Processors, 1990. ICCD'90. Proceedings., 1990 IEEE International Conference on*, pages 220–223. IEEE, 1990.

[24] A. Mader. Modal μ-calculus, model checking and gauß elimination. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 72–88, 1995.

[25] A. Mader. *Verification of modal properties using boolean equation systems.* Bertz Verlag, 1997.

[26] S. Orzan, W. Wesselink, and T. Willemse. Static analysis techniques for parameterised boolean equation systems. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 230–245, 2009.

[27] P. Stevens and C. Stirling. Practical model-checking using games. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 85–101, 1998.

[28] C. Stirling. Modal and temporal logics. In *Handbook of logic in computer science (vol. 2)*, pages 477–563. Oxford University Press, Inc., 1993.

[29] A. van Dam, B. Ploeger, and T. Willemse. Instantiation for parameterised boolean equation systems. *Theoretical Aspects of Computing-ICTAC 2008*, pages 440–454, 2008.

[30] M Van Eekelen, S. Ten Hoedt, R. Schreurs, and Y. Usenko. Analysis of a session-layer protocol in mcrl2. *Formal Methods for Industrial Critical Systems*, pages 182–199, 2008.

[31] B. Vergauwen and J. Lewi. A linear algorithm for solving fixed-point equations on transition systems. *CAAP'92*, pages 322–341, 1992.

[32] W. Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theoretical Computer Science*, 200(1-2):135–183, 1998.

A Coq

The forms, transformations and proofs from Section 3.5 were also performed using the proof assistant software Coq[5]. Doing these proofs in a proof assistant gives us more certainty of the validity of the proved lemmas and theorems. In addition to this, proving properties on PBESs using this framework results in useful feedback for further development of the PBES formalization in Coq.

In this Appendix, we give listings of the implementations in Coq.

A.1 PBES Framework

The proofs make use of a framework around PBESs in Coq, developed by Carst Tankink[6]. The proofs are done on an experimental version of this framework. A cleaned-up version of the used framework is given below.

```
1  (** This structure/record describes a language, parameterized over a type of 'names' and an
       interpretation of these names to actual Coq terms. *)
2  Structure language := {
3    (** Types for our language *)
4    type : Type
5    (** Denotation takes the abstract names in type, and interprets them into Coq datatypes. *)
6  ; type_denote : type -> Type
7
8  (** Operators. Just binary for now. Unary, ternary etc can be added here when necessary. *)
9    (** We declare a set of names, together with the intended signature. *)
10 ; binop: type -> type -> type -> Type
11 (** Just like with type, the binary operations takes a name plus a signature, and interprets it
       to an actual Coq operator. *)
12 ; binop_denote : forall t1 t2 t3, binop t1 t2 t3 -> type_denote t1 -> type_denote t2 ->
       type_denote t3
13 }.
14
15 (** Set some implicit arguments. *)
16 Arguments type_denote {_} _ . (* The language we work on can be determined from the type given.
       *)
17 Arguments binop {_} _ _ _ . (* The language is implicit. Name and signature are required. *)
18 Arguments binop_denote {_ _ _ _} _ _ _. (* Given the operator, we can derive the signature. You
       need to give an interpretation of the signature, though. *)
19
20 (** We now define a simple language, over naturals and propositions, with a few operators. To
       add to the language (provided you do not want to add another arity), you do it here.*)
21
22 Require Import Program.
23
24 (** The language ranges over a type of 'nats' and a type of 'propositions' *)
25 Inductive sort := nat_sort | prop_sort.
26
27 (** We take the natural interpretation for the sorts. *)
28 Definition sort_denote s: Type :=
29   match s with
30   | nat_sort => nat
31   | prop_sort => Prop
32   end.
33
34 (** Binary operators are defined similarly: they have a symbol and a signature. *)
35 Inductive sort_binop: sort -> sort -> sort -> Type :=
36   (** Pure binary operators. *)
37   | and_symb  : sort_binop prop_sort prop_sort prop_sort
38   | or_symb   : sort_binop prop_sort prop_sort prop_sort
39   | impl_symb : sort_binop prop_sort prop_sort prop_sort
40   (** Relation between naturals: equality. *)
41   | eq_symb   : sort_binop nat_sort  nat_sort  prop_sort
42
43   (** Binary functions *)
44   | plus_symb : sort_binop nat_sort  nat_sort  nat_sort
45   | mult_symb : sort_binop nat_sort  nat_sort  nat_sort.
46
47 (** The interpretation is as expected. *)
48 Definition sort_binop_denote {s1 s2 s3} (op : sort_binop s1 s2 s3):
49     sort_denote s1 -> sort_denote s2 -> sort_denote s3 :=
```

[5]Coq Proof Assistant (http://coq.inria.fr/), version 8.4rc1, running on a 64 bits GNU/Linux machine.
[6]http://www.cs.ru.nl/~carst/

```
50   match op with
51   | and_symb => and
52   | or_symb  => or
53   | impl_symb => impl
54   (** @eq is Leibniz equality *)
55   | eq_symb  => @eq nat
56   | plus_symb => plus
57   | mult_symb => mult
58   end.
59
60 (** Given the building blocks, we define the language of data sorts as being Canonical,
     allowing Coq to do a bit more type inference. *)
61 Canonical Structure sorts := {|
62   type := sort
63 ; type_denote := sort_denote
64 ; binop := sort_binop
65 ; binop_denote := @sort_binop_denote
66 |}.
67
68 (** Given a language, we can define expressions over that language. *)
69 Section data_exp.
70 Variable l : language.
71
72 (** Right now, we use arbitrary naturals for data variable names. This is not (yet) de Bruijn
     indices, but could be extended to be so when necessary. *)
73 Definition  var := nat.
74
75 (** Note that data expressions are typed (with their expected result). *)
76 Inductive data_exp  : type l -> Type :=
77 (** Variables take a type argument and a name. *)
78  | d_var   t : var -> data_exp t
79 (** Constants are taken from the actual types, and fed into the data_exp type. *)
80  | d_const t : type_denote t -> data_exp t
81 (** Building up expresions out of binary operators. Note: when adding other arities above, add
     an inhabitant of data_exp here as well. *)
82  | d_binop t1 t2 t3 : forall op: binop t1 t2 t3, data_exp t1 -> data_exp t2 -> data_exp t3.
83
84 (** Data_env gives values to variables. They are stratified by type, so you have a set of names
      for nat_sort and one for prop_sort. An alternative would be to type the variables
     themselves, and have environment rely on that type. I am not yet sure which is better. *)
85 Definition data_env := forall t:  type l, var -> type_denote t.
86
87 (** Data_evaluation takes a data expression and a data environment and evaluates the expression
       to a Coq expression *)
88 Definition data_eval (E: data_env): forall t, data_exp t -> type_denote t :=
89 fix eval {t} (e: data_exp t) :=
90   match e with
91   | d_var t v      => E t v
92   | d_const t c => c
93   | d_binop t1 t2 t3 op e1 e2 => binop_denote op (eval e1) (eval e2)
94   end.
95
96 End data_exp.
97
98 Arguments data_eval {_} _ {_} _.
99 Arguments d_binop {_ _ _ _} _ _ _.
100 Arguments d_var {_} _ _.
101 Arguments d_const {_} _ {_}.
102
103 (** Right now, recursion variables are just names. *)
104 Definition recvar := nat.
105
106 (** Predicate formulae *)
107 Inductive pred_form :=
108   (** First, the constants: true and false. *)
109   | p_true    : pred_form
110   | p_false   : pred_form
111
112   (** Data expressions that result in propositions can be used as part of a predicate formula.
       *)
113   | p_bool    : data_exp _ prop_sort -> pred_form
114
115   (** Recursion variables occur in the predicate formulae. They are a name together with a data
        expression of the correct type. *)
116   | p_var     : forall s: type sorts, recvar -> data_exp sorts s  -> pred_form
117
118 (** Binary connectors for predicate formulae. *)
119   | p_and     : pred_form -> pred_form -> pred_form
120   | p_or      : pred_form -> pred_form -> pred_form
121
```

```
122 (** Quantification. *)
123   | p_forall : var -> type sorts -> pred_form -> pred_form
124   | p_exists : var -> type sorts -> pred_form -> pred_form.
125
126 (** Standard environment for interpreting recursion variables. *)
127 Definition recvar_env := forall s: type sorts, recvar -> (type_denote s -> Prop).
128
129 (** Boolean equality on our own sorts. *)
130 Definition beq_sort (t1 t2: sort) : bool :=
131   match t1, t2 with
132   | nat_sort, nat_sort => true
133   | prop_sort, prop_sort => true
134   | _, _ => false
135   end.
136
137 (** Boolean equality can be used to establish Leibniz equality. Just run the algorithm. *)
138 Lemma beq_eq_conv (t1 t2: sort) : beq_sort t1 t2 = true -> t1 = t2.
139 Proof.
140   intros t1t2.
141   destruct t1, t2.
142     reflexivity.
143     simpl in t1t2. inversion t1t2.
144     simpl in t1t2. inversion t1t2.
145     reflexivity.
146 Qed.
147
148 Require Import EqNat.
149
150 (** Update on data environements, defined interactively (there is some type conversion going on
      )
151     Update takes an environment 'eps' a type and a value of that type, and a variable.
152     It returns an environment that returns value for x, and the value in 'eps' for any other
          variable. *)
153 Definition update (eps: data_env sorts) {t: sort} (value : type_denote t) (x: var): data_env
      sorts.
154 Proof.
155   unfold data_env.
156
157 (** The new data environment is a function that takes a type and a variable. (with the intended
        relation x':t' *)
158   intros t' x'.
159 (** Deternine whether x and x' have the same type, are the same. the _eqn keeps the equations
      in the context. *)
160   destruct (beq_sort t t') as []_eqn.
161   * destruct (beq_nat x x') as []_eqn.
162     - rewrite <- beq_eq_conv with (t1 := t) by assumption. (** x' = x, t' = t. Return value. *)
163       apply value.
164     - apply eps. apply x'. (** Types match, find it in eps. *)
165   * apply eps. apply x'. (** When the type don't match, don't bother with the variable
          comparison, lookup in eps. *)
166 Defined.
167
168 (** Semantics for predicate formulae: creates Coq propositions/predicates. *)
169 Fixpoint pred_sem (p: pred_form) (eta: recvar_env) (eps: data_env sorts): Prop :=
170   match p with
171   | p_true => True
172   | p_false => False
173   (** Evaluate data expressions in the given environment. *)
174   | p_bool exp => data_eval eps exp
175   (** Recursion variables evaluate to predicates over data. *)
176   | p_var s name exp => eta s name (data_eval eps exp)
177   | p_and phi1 phi2 => pred_sem phi1 eta eps /\ pred_sem phi2 eta eps
178   | p_or phi1 phi2 => pred_sem phi1 eta eps \/ pred_sem phi2 eta eps
179   (** Quantifications become quantifications on the Coq level, over the actual Coq types. *)
180   | p_forall d D phi => forall x: type_denote D, pred_sem phi eta (update eps x d)
181   | p_exists d D phi => exists x: type_denote D, pred_sem phi eta (update eps x d)
182   end.
```

A.2 Strongly Guarded Form

First, some helper definitions are done:

```
1 Require Import Bool.
2
3 (** Whether p is a (possibly generalized) conjunction. **)
4 Definition is_conjunction(p: pred_form) : bool :=
5   match p with
```

```
6     | p_true => false
7     | p_false => false
8     | p_bool b => false
9     | p_var _ _ _ => false
10    | p_and _ _ => true
11    | p_or _ _ => false
12    | p_forall _ _ _ => true
13    | p_exists _ _ _ => false
14  end.
15
16  (** Whether p is a (possibly generalized) disjunction. **)
17  Definition is_disjunction(p: pred_form) : bool :=
18    match p with
19    | p_true => false
20    | p_false => false
21    | p_bool b => false
22    | p_var _ _ _ => false
23    | p_and _ _ => false
24    | p_or _ _ => true
25    | p_forall _ _ _ => false
26    | p_exists _ _ _ => true
27  end.
28
29  (** Whether a predicate formula is a simple boolean formula (recursion variable free). **)
30  Fixpoint is_bool_expr(p: pred_form) : Prop :=
31    match p with
32    | p_true => True
33    | p_false => True
34    | p_bool b => True
35    | p_var _ _ _ => False
36    | p_and phi1 phi2 => (is_bool_expr phi1) /\ (is_bool_expr phi2)
37    | p_or phi1 phi2 => (is_bool_expr phi1) /\ (is_bool_expr phi2)
38    | p_forall d D phi => (is_bool_expr phi)
39    | p_exists d D phi => (is_bool_expr phi)
40  end.
41  Fixpoint is_bool_expr_b(p: pred_form) : bool :=
42    match p with
43    | p_true => true
44    | p_false => true
45    | p_bool b => true
46    | p_var _ _ _ => false
47    | p_and phi1 phi2 => (is_bool_expr_b phi1) && (is_bool_expr_b phi2)
48    | p_or phi1 phi2 => (is_bool_expr_b phi1) && (is_bool_expr_b phi2)
49    | p_forall d D phi => (is_bool_expr_b phi)
50    | p_exists d D phi => (is_bool_expr_b phi)
51  end.
52
53  (** Small lemma to help combine is_bool_expr and is_bool_expr_b. **)
54  Lemma is_bool_expr_bool (p: pred_form):
55    is_bool_expr_b p = true <-> is_bool_expr p.
56  Proof.
57  induction p; simpl; try tauto.
58  split. intros. apply eq_true_false_abs with true. tauto. rewrite H. tauto. contradiction.
59  split; intros. rewrite andb_true_iff in H. tauto.
60  rewrite andb_true_iff. tauto.
61  split; intros. rewrite andb_true_iff in H. tauto.
62  rewrite andb_true_iff. tauto.
63  Qed.
```

Definition 3.4

```
1   Fixpoint guard_and(p: pred_form) : pred_form :=
2     match p with
3     | p_true => p_true
4     | p_false => p_false
5     | p_bool b => p_bool b
6     | p_var _ _ _ => p_true
7     | p_and phi1 phi2 => p_and (guard_and phi1) (guard_and phi2)
8     | p_or phi1 phi2 => p_or (guard_and phi1) (guard_and phi2)
9     | p_forall d D phi => p_forall d D (guard_and phi)
10    | p_exists d D phi => p_exists d D (guard_and phi)
11  end.
12  Fixpoint guard_or(p: pred_form) : pred_form :=
13    match p with
14    | p_true => p_true
15    | p_false => p_false
16    | p_bool b => p_bool b
17    | p_var _ _ _ => p_false
```

```
18    | p_and phi1 phi2 => p_and (guard_or phi1) (guard_or phi2)
19    | p_or phi1 phi2 => p_or (guard_or phi1) (guard_or phi2)
20    | p_forall d D phi => p_forall d D (guard_or phi)
21    | p_exists d D phi => p_exists d D (guard_or phi)
22  end.
```

We define some helper Lemmas about the results of the guard transformations:

```
1  (** The results of guard_and and guard_or are always simple booleans expressions. **)
2  Lemma guard_bool_expr (p: pred_form):
3    is_bool_expr (guard_and p) /\ is_bool_expr (guard_or p).
4  Proof.
5  induction p; simpl; tauto.
6  Qed.
7
8  (** If a guard_function is used on a simple boolean expression, it acts as identity. **)
9  Lemma guard_or_on_bool_expr (p: pred_form):
10   is_bool_expr p -> (guard_or p) = p.
11 Proof.
12 intros. induction p; simpl; simpl in H; try tauto.
13   (** Case p_and **)
14   assert (guard_or p1 = p1). apply IHp1. apply H.
15   assert (guard_or p2 = p2). apply IHp2. apply H.
16   rewrite H0, H1. tauto.
17   (** Case p_or **)
18   assert (guard_or p1 = p1). apply IHp1. apply H.
19   assert (guard_or p2 = p2). apply IHp2. apply H.
20   rewrite H0, H1. tauto.
21   (** Case p_exists **)
22   assert (guard_or p = p). apply IHp. apply H.
23   rewrite H0. tauto.
24   (** Case p_exists **)
25   assert (guard_or p = p). apply IHp. apply H.
26   rewrite H0. tauto.
27 Qed.
28 Lemma guard_and_on_bool_expr (p: pred_form):
29   is_bool_expr p -> (guard_and p) = p.
30 Proof.
31 intros. induction p; simpl; simpl in H; try tauto.
32   (** Case p_and **)
33   assert (guard_and p1 = p1). apply IHp1. apply H.
34   assert (guard_and p2 = p2). apply IHp2. apply H.
35   rewrite H0, H1. tauto.
36   (** Case p_or **)
37   assert (guard_and p1 = p1). apply IHp1. apply H.
38   assert (guard_and p2 = p2). apply IHp2. apply H.
39   rewrite H0, H1. tauto.
40   (** Case p_exists **)
41   assert (guard_and p = p). apply IHp. apply H.
42   rewrite H0. tauto.
43   (** Case p_exists **)
44   assert (guard_and p = p). apply IHp. apply H.
45   rewrite H0. tauto.
46 Qed.
47
48 (** guard_X . guard_Y = guard_Y **)
49 Lemma guard_guard (p : pred_form):
50   ((guard_and (guard_and p)) = (guard_and p)) /\
51   ((guard_and (guard_or p)) = (guard_or p)) /\
52   ((guard_or (guard_and p)) = (guard_and p)) /\
53   ((guard_or (guard_or p)) = (guard_or p)).
54 Proof.
55 induction p; simpl; repeat split; intuition;
56 try rewrite H;
57 try rewrite H0; try rewrite H1; try rewrite H2; try rewrite H3;
58 try rewrite H4; try rewrite H5; try rewrite H6; try rewrite H7;
59 tauto.
60 Qed.
```

Lemma 3.5

```
1  Lemma guard_strength (p : pred_form) :
2    forall eta1 eta2 eps, (
3      (pred_sem p eta1 eps -> pred_sem (guard_and p) eta2 eps) /\
4      (pred_sem (guard_or p) eta1 eps -> pred_sem p eta2 eps)
5    ).
6  Proof.
7  intro. intro.
```

```
8  induction p; simpl; try tauto.
9    (* Case p_and *)
10   split; intros; destruct H; split.
11     apply IHp1. apply H.
12     apply IHp2. apply H0.
13     apply IHp1. apply H.
14     apply IHp2. apply H0.
15   (* Case p_or *)
16   split; intros; destruct H.
17     left. apply IHp1. apply H.
18     right. apply IHp2. apply H.
19     left. apply IHp1. apply H.
20     right. apply IHp2. apply H.
21   (* Case p_forall *)
22   split; intros; apply IHp; apply H.
23   (* Case p_exists *)
24   split; intros; destruct H; exists x; apply IHp; apply H.
25 Qed.
```

Lemma 3.6

```
1  Lemma guard (p : pred_form) :
2    (forall eta eps, ((pred_sem (guard_and p) eta eps) <-> (pred_sem p (fun (so: type sorts) (r:
         recvar) (t: type_denote so) => True) eps)) /\
3                    ((pred_sem (guard_or p) eta eps) <-> (pred_sem p (fun (so: type sorts) (r:
                         recvar) (t: type_denote so) => False) eps))).
4  Proof.
5  simpl. induction p; intros; simpl; try tauto.
6    (* Case and *)
7    destruct (IHp1 eta eps), (IHp2 eta eps).
8    split; split; split; tauto.
9    (* Case or *)
10   destruct (IHp1 eta eps), (IHp2 eta eps).
11   split; split; intros; tauto.
12   (* Case forall *)
13   split; split; intros; destruct (IHp eta (update eps x v)); intuition.
14   (* Case exists *)
15   split; split; intros; destruct H; exists x; destruct (IHp eta (update eps x v)); intuition.
16 Qed.
17
18 Lemma guard_strength2 (p : pred_form) :
19   (forall p',
20     (forall eta eta' eps, (pred_sem p eta eps) -> (pred_sem p' eta' eps)) ->
21     (forall eta'' eps'', (pred_sem (guard_and p) eta'' eps'') -> (pred_sem p' eta'' eps''))
22   ) /\
23   (forall p',
24     (forall eta eta' eps, (pred_sem p' eta eps) -> (pred_sem p eta' eps)) ->
25     (forall eta'' eps'', (pred_sem p' eta'' eps'') -> (pred_sem (guard_or p) eta'' eps''))
26   ).
27 Proof.
28 split; intros.
29 apply H with (fun (so: type sorts) (r: recvar) (t: type_denote so) => True).
30 apply (guard p eta'' eps''). apply H0.
31
32 apply (guard p eta'' eps'').
33 apply H with eta''. apply H0.
34 Qed.
```

Definition 3.8 Note that these transformations are slightly different, in order to make them trivially well-defined.

```
1  Fixpoint F_or(p: pred_form) : pred_form :=
2    if (is_bool_expr_b p) then p_false else
3    match p with
4    | p_true => p_false
5    | p_false => p_false
6    | p_bool _ => p_false
7    | p_var s name expr => p_var s name expr
8    | p_and phi1 phi2 => p_and (guard_and (p_and phi1 phi2)) (if (is_bool_expr_b phi1) then (
         F_and phi2) else if (is_bool_expr_b phi2) then (F_and phi1) else (p_and (F_and phi1) (
         F_and phi2)))
9    | p_or phi1 phi2 => if (is_bool_expr_b phi1) then (F_or phi2) else if (is_bool_expr_b phi2)
         then (F_or phi1) else (p_or (F_or phi1) (F_or phi2))
10   | p_forall d D phi => p_and (guard_and (p_forall d D phi)) (p_forall d D (p_or (guard_or phi)
          (F_or phi)))
11   | p_exists d D phi => p_exists d D (p_and (guard_and phi) (F_and phi))
```

```
12 end
13 with F_and(p: pred_form) : pred_form :=
14   if (is_bool_expr_b p) then p_true else
15   match p with
16   | p_true => p_true
17   | p_false => p_true
18   | p_bool _ => p_true
19   | p_var s name expr => p_var s name expr
20   | p_and phi1 phi2 => if (is_bool_expr_b phi1) then (F_and phi2) else if (is_bool_expr_b phi2)
       then (F_and phi1) else (p_and (F_and phi1) (F_and phi2))
21   | p_or phi1 phi2 => p_or (guard_or (p_or phi1 phi2)) ((if (is_bool_expr_b phi1) then (F_or
       phi2) else if (is_bool_expr_b phi2) then (F_or phi1) else (p_or (F_or phi1) (F_or phi2))
       ))
22   | p_forall d D phi => p_forall d D (p_or (guard_or phi) (F_or phi))
23   | p_exists d D phi => p_or (guard_or (p_exists d D phi)) (p_exists d D (p_and (guard_and phi)
       (F_and phi)))
24 end.
25
26 (** Right-hand side of F applied to a complete equation. **)
27 Definition rhs(p: pred_form) : pred_form :=
28   (if is_conjunction p then p_and (guard_and p) (F_and p) else p_or (guard_or p) (F_or p)).
```

Lemma 3.10 and Theorem 3.11

```
1  Theorem sound (p: pred_form):
2    forall eta eps, pred_sem (rhs p) eta eps <-> pred_sem p eta eps.
3  Proof.
4
5  (** Case distinction without splitting the induction hypotheses. **)
6  assert (forall eta eps,
7    (pred_sem (p_and (guard_and p) (F_and p)) eta eps <-> pred_sem p eta eps) /\
8    (pred_sem (p_or (guard_or p) (F_or p)) eta eps <-> pred_sem p eta eps)).
9
10 induction p; simpl; try tauto; intros.
11   (** Case p_and **)
12   destruct (IHp1 eta eps), (IHp2 eta eps).
13   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros.
14     (** p1 and p2 are simple boolean expressions. **)
15     rewrite is_bool_expr_bool in H3, H4.
16     repeat rewrite guard_and_on_bool_expr.
17     repeat rewrite guard_or_on_bool_expr.
18     simpl. tauto. tauto. tauto. tauto. tauto.
19     (** p1 is simple boolean expression, p2 is not. **)
20     rewrite is_bool_expr_bool in H4. simpl.
21     repeat rewrite guard_and_on_bool_expr with p1.
22     repeat rewrite guard_or_on_bool_expr with p1.
23     split; intros. rewrite<- H1. simpl. tauto.
24     simpl. split; intros. destruct H5.
25     rewrite<- H2. simpl. tauto. rewrite<- H1. simpl. tauto.
26     right. rewrite<- H1 in H5. simpl in H5. tauto.
27     tauto. tauto.
28     (** p2 is simple boolean expression, p1 is not. **)
29     rewrite is_bool_expr_bool in H3. simpl.
30     repeat rewrite guard_and_on_bool_expr with p2.
31     repeat rewrite guard_or_on_bool_expr with p2.
32     split; intros. rewrite<- H. simpl. tauto.
33     split; intros. destruct H5.
34     rewrite<- H0. simpl. tauto. rewrite<- H. simpl. tauto.
35     right. rewrite<- H in H5. simpl in H5. tauto.
36     tauto. tauto.
37     (** neither are simple boolean expressions. **)
38     simpl. split; split; intros.
39       rewrite<- H,<- H1. simpl. tauto.
40       rewrite<- H,<- H1 in H5. simpl in H5. tauto.
41       destruct H5.
42       rewrite<- H0,<- H2. simpl. tauto.
43       rewrite<- H,<- H1. simpl. tauto.
44       rewrite<- H,<- H1 in H5. simpl in H5. tauto.
45   (** Case p_or **)
46   destruct (IHp1 eta eps), (IHp2 eta eps).
47   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros.
48     (** p1 and p2 are simple boolean expressions. **)
49     rewrite is_bool_expr_bool in H3, H4.
50     repeat rewrite guard_and_on_bool_expr.
51     repeat rewrite guard_or_on_bool_expr.
52     simpl. tauto. tauto. tauto. tauto. tauto.
53     (** p1 is simple boolean expression, p2 is not. **)
54     rewrite is_bool_expr_bool in H4. simpl.
```

```
55      repeat rewrite guard_and_on_bool_expr with p1.
56      repeat rewrite guard_or_on_bool_expr with p1.
57      split; intros. split; intros. rewrite<- H2. simpl. tauto.
58      split. rewrite<- H1 in H5. simpl in H5. tauto. rewrite<- H2 in H5. simpl in H5. tauto.
59      rewrite<- H2. simpl. tauto. tauto. tauto.
60      (** p2 is simple boolean expression, p1 is not. **)
61      rewrite is_bool_expr_bool in H3. simpl.
62      repeat rewrite guard_and_on_bool_expr with p2.
63      repeat rewrite guard_or_on_bool_expr with p2.
64      split; intros. split; intros. rewrite<- H0. simpl. tauto.
65      split. rewrite<- H in H5. simpl in H5. tauto. rewrite<- H0 in H5. simpl in H5. tauto.
66      rewrite<- H0. simpl. tauto. tauto.
67      (** neither are simple boolean expressions. **)
68      simpl. split; split; intros.
69        rewrite<- H0,<- H2. simpl. tauto.
70        split. rewrite<- H,<- H1 in H5. simpl in H5. tauto.
71        rewrite<- H0,<- H2 in H5. simpl in H5. tauto.
72        rewrite<- H0,<- H2. simpl. tauto.
73        rewrite<- H0,<- H2 in H5. simpl in H5. tauto.
74    (** Case p_forall **)
75    case_eq (is_bool_expr_b p); intros; simpl.
76      (** p is a simple boolean expression. **)
77      rewrite guard_and_on_bool_expr. rewrite guard_or_on_bool_expr. tauto.
78      apply is_bool_expr_bool. apply H. apply is_bool_expr_bool. apply H.
79      (** p is no simple boolean expression. **)
80      split; split; intros. destruct (IHp eta (update eps x v)). apply H2. apply H0.
81      split; intros; destruct (IHp eta (update eps x v)). apply H1. apply H0.
82      apply H2. apply H0.
83      destruct (IHp eta (update eps x v)). destruct H0. apply H2. left. apply H0.
84      destruct H0. destruct (H3 x).
85      apply H2. left. apply H4. apply H1. split. apply H0. apply H1. apply H2. right. apply H4.
86      right. split; intros; destruct (IHp eta (update eps x v)). apply H1. apply H0.
87      apply H2. apply H0.
88    (** Case p_exists **)
89    case_eq (is_bool_expr_b p); intros; simpl.
90      (** p is a simple boolean expression. **)
91      rewrite guard_and_on_bool_expr. rewrite guard_or_on_bool_expr. tauto.
92      apply is_bool_expr_bool. apply H. apply is_bool_expr_bool. apply H.
93      (** p is no simple boolean expression. **)
94      split. split; intros. destruct H0.
95      destruct H1; destruct H1; exists x; destruct (IHp eta (update eps x v)).
96      apply H3. left. apply H1. apply H3. apply H3.
97      destruct H0. apply H2. apply H1.
98      destruct H0. split. exists x. destruct (IHp eta (update eps x v)). apply H1. apply H0.
99      destruct (IHp eta (update eps x v)). rewrite<- H2 in H0. destruct H0.
100     left. exists x. apply H0. right. exists x. split. apply H1. apply H2. apply H2.
101     apply H2. right. apply H0.
102     apply H1. apply H2. right. apply H0.
103     split; intros. destruct H0. destruct H0. exists x. destruct (IHp eta (update eps x v)).
104     apply H2. left. apply H0.
105     destruct H0. exists x. destruct (IHp eta (update eps x v)). apply H1. apply H0.
106     destruct H0. right. exists x. destruct (IHp eta (update eps x v)). apply H1. apply H0.
107
108   unfold rhs. case (is_conjunction p); intros; destruct (H eta eps); tauto.
109 Qed.
```

Definition 3.7

```
1  (** Check whether en equation is a guarded conjunction.
2  This checks whether phi1 is a valid guard and the conjunction of p1 and p2 satisfies the
       strongly guarded property. **)
3  Definition is_guarded_conj(phi1: pred_form)(phi2: pred_form) : Prop :=
4  (is_bool_expr phi1) /\ (forall eta eps, (pred_sem phi1 eta eps) -> (pred_sem (p_and phi1 (
       guard_and phi2)) eta eps)).
5
6  (** Check whether en equation is a guarded disjunction.
7  This checks whether phi1 is a valid guard and the disjunction of p1 and p2 satisfies the
       strongly guarded property. **)
8  Definition is_guarded_disj(phi1: pred_form)(phi2: pred_form) : Prop :=
9  (is_bool_expr phi1) /\ (forall eta eps, (pred_sem (p_or phi1 (guard_or phi2)) eta eps) -> (
       pred_sem phi1 eta eps)).
10
11 (** Whether a predicate formula is of the form RHS_and / RHS_or exactly.
12 This is very strong: there is no leniency for reversed parameters to operators and removed
       trivial subexpressions. **)
13 Fixpoint is_rhs_and(p: pred_form) : Prop :=
14   match p with
15   | p_true => False
```

```
16   | p_false => False
17   | p_bool b => False
18   | p_var _ _ _ => True
19   | p_and phi1 phi2 => (is_rhs_and phi1) /\ (is_rhs_and phi2)
20   | p_or phi1 phi2 => (is_guarded_disj phi1 phi2) /\ (is_rhs_or phi2)
21   | p_forall d D phi => (is_forall_body phi)
22   | p_exists d D phi => False
23 end
24 with is_rhs_or(p: pred_form) : Prop :=
25   match p with
26   | p_true => False
27   | p_false => False
28   | p_bool b => False
29   | p_var _ _ _ => True
30   | p_and phi1 phi2 => (is_guarded_conj phi1 phi2) /\ (is_rhs_and phi2)
31   | p_or phi1 phi2 => (is_rhs_or phi1) /\ (is_rhs_or phi2)
32   | p_forall d D phi => False
33   | p_exists d D phi => (is_exists_body phi)
34 end
35
36 (** Whether the predicate formula is of the form required in the body of a universal /
       existential quantifier. **)
37 with is_forall_body(p: pred_form) : Prop :=
38   match p with
39   | p_or phi1 phi2 => (is_guarded_disj phi1 phi2) /\ (is_rhs_or phi2)
40   | _ => False
41 end
42 with is_exists_body(p: pred_form) : Prop :=
43   match p with
44   | p_and phi1 phi2 => (is_guarded_conj phi1 phi2) /\ (is_rhs_and phi2)
45   | _ => False
46 end.
47
48 (** Whether the predicate formula is of the form RHS. **)
49 Definition is_rhs(p: pred_form) : Prop :=
50   (is_rhs_and p) \/ (is_rhs_or p) \/ (is_bool_expr p).
```

Lemma 3.12

```
1  (** A lemma which helps in the proof of guard_F. **)
2  Lemma guard_F2 (p : pred_form):
3    (forall eta eps,
4      (((pred_sem (guard_and p) eta eps) -> (pred_sem (guard_and (F_and p)) eta eps)) /\
5      ((pred_sem (guard_or (F_or p)) eta eps) -> (pred_sem (guard_or p) eta eps))) ->
6      (((pred_sem (guard_and p) eta eps) -> (pred_sem (guard_and (p_or (guard_or p) (F_or p)))
           eta eps)) /\
7      ((pred_sem (guard_or (p_and (guard_and p) (F_and p))) eta eps) -> (pred_sem (guard_or p)
           eta eps)))).
8  Proof.
9  intros.
10 induction p; simpl; try tauto.
11   (** Case p_and **)
12   assert (forall p: pred_form, (guard_and (guard_and p)) = (guard_and p)) as
         guard_and_guard_and. intros. apply guard_guard.
13   split; intros; simpl in H; destruct H;
14   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros;
15   simpl; rewrite H2, H3 in H, H1; simpl in H, H1; try rewrite H2, H3 in H0; simpl in H0;
16   try rewrite is_bool_expr_bool in H2; try rewrite is_bool_expr_bool in H3.
17     (** p1 and p2 are simple boolean expressions. **)
18     left. rewrite (guard_or_on_bool_expr p1), (guard_or_on_bool_expr p2). apply H0. apply H2.
           apply H3.
19     right. split. rewrite (guard_and_guard_and p1), (guard_and_guard_and p2). apply H0. apply H
           . apply H0.
20     right. split. rewrite (guard_and_guard_and p1), (guard_and_guard_and p2). apply H0. apply H
           . apply H0.
21     right. split. rewrite (guard_and_guard_and p1), (guard_and_guard_and p2). apply H0. apply H
           . apply H0.
22     rewrite (guard_and_on_bool_expr p1), (guard_and_on_bool_expr p2) in H0. apply H0. apply H2.
            apply H3.
23     (** p1 is simple boolean expression, p2 is not. **)
24     split. rewrite (guard_and_on_bool_expr p1) in H0. apply H0. apply H3. apply H1. apply H0.
25     (** p2 is simple boolean expression, p1 is not. **)
26     split. apply H1. apply H0.
27     rewrite (guard_and_on_bool_expr p2) in H0. apply H0. apply H2.
28     (** neither are simple boolean expressions. **)
29     apply H1. apply H0.
30   (** Case p_or **)
31   assert (forall p: pred_form, (guard_or (guard_or p)) = (guard_or p)) as guard_or_guard_or.
         intros. apply guard_guard.
```

```
32   split; intros; simpl in H; destruct H;
33   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros;
34   simpl; rewrite H2, H3 in H, H1; simpl in H, H1; try rewrite H2, H3 in H0; simpl in H0;
35   try rewrite is_bool_expr_bool in H2; try rewrite is_bool_expr_bool in H3.
36     (** p1 and p2 are simple boolean expressions. **)
37     rewrite (guard_or_on_bool_expr p1), (guard_or_on_bool_expr p2). left. apply H0. apply H2.
         apply H3.
38     apply H. apply H0. apply H. apply H0. apply H. apply H0.
39     rewrite (guard_and_on_bool_expr p1), (guard_and_on_bool_expr p2) in H0. apply H0. apply H2.
          apply H3.
40     (** p1 is simple boolean expression, p2 is not. **)
41     destruct H0. destruct H4.
42     rewrite guard_or_guard_or, guard_or_guard_or in H4. apply H4.
43     apply H1. apply H4.
44     (** p2 is simple boolean expression, p1 is not. **)
45     destruct H0. destruct H4.
46     rewrite guard_or_guard_or, guard_or_guard_or in H4. apply H4.
47     apply H1. apply H4.
48     (** neither are simple boolean expressions. **)
49     destruct H0. destruct H4.
50     rewrite guard_or_guard_or, guard_or_guard_or in H4. apply H4.
51     apply H1. apply H4.
52   (** Case p_forall **)
53   assert (forall p: pred_form, (guard_and (guard_and p)) = (guard_and p)) as
       guard_and_guard_and. intros. apply guard_guard.
54   split; intros; simpl in H;
55   case_eq (is_bool_expr_b p); intros; simpl; rewrite H1 in H; simpl in H; try rewrite H1 in H0;
        simpl in H0.
56     (** p is a simple boolean expression. **)
57     rewrite is_bool_expr_bool in H1.
58     rewrite (guard_or_on_bool_expr p). left. apply H0. apply H1.
59     (** p is no simple boolean expression. **)
60     right. split. rewrite guard_and_guard_and. apply H0.
61     apply H. apply H0.
62     (** p is a simple boolean expression. **)
63     rewrite is_bool_expr_bool in H1.
64     rewrite (guard_and_on_bool_expr p) in H0. apply H0. apply H1.
65     (** p is no simple boolean expression. **)
66     apply H. apply H0.
67   (** Case p_exists **)
68   assert (forall p: pred_form, (guard_or (guard_or p)) = (guard_or p)) as guard_or_guard_or.
       intros. apply guard_guard.
69   split; intros; simpl in H;
70   case_eq (is_bool_expr_b p); intros; simpl; rewrite H1 in H; simpl in H; try rewrite H1 in H0;
        simpl in H0.
71     (** p is a simple boolean expression. **)
72     rewrite is_bool_expr_bool in H1.
73     rewrite guard_or_on_bool_expr. left. apply H0. apply H1.
74     (** p is no simple boolean expression. **)
75     destruct H. destruct H. apply H0. left. apply H. right. apply H.
76     (** p is a simple boolean expression. **)
77     rewrite is_bool_expr_bool in H1.
78     rewrite guard_and_on_bool_expr in H0. apply H0. apply H1.
79     (** p is no simple boolean expression. **)
80     destruct H0. destruct H2. rewrite guard_or_guard_or in H2. apply H2.
81     apply H. apply H2.
82 Qed.
83
84 (** Used for proving strongly guardedness. **)
85 Lemma guard_F (p : pred_form):
86   forall eta eps,
87     ((pred_sem (guard_and p) eta eps) -> (pred_sem (guard_and (F_and p)) eta eps)) /\
88     ((pred_sem (guard_or (F_or p)) eta eps) -> (pred_sem (guard_or p) eta eps)).
89 Proof.
90 induction p; intros; simpl; try tauto.
91   (** Case p_and **)
92   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros; simpl;
93   try rewrite is_bool_expr_bool in H; try rewrite is_bool_expr_bool in H0.
94     (** p1 and p2 are simple boolean expressions. **)
95     tauto.
96     (** p1 is simple boolean expression, p2 is not. **)
97     split; intros. apply IHp2. apply H1.
98     split. rewrite (guard_and_on_bool_expr p1) in H1. apply H1. apply H0.
99     apply guard_F2. split; intros; apply IHp2; apply H2. simpl. tauto.
100    (** p2 is simple boolean expression, p1 is not. **)
101    split; intros. apply IHp1. apply H1.
102    split. apply guard_F2. split; intros; apply IHp1; apply H2. simpl. tauto.
103    rewrite (guard_and_on_bool_expr p2) in H1. apply H1. apply H.
104    (** neither are simple boolean expressions. **)
105    split; intros. split. apply IHp1. apply H1. apply IHp2. apply H1.
```

```
106     split; apply guard_F2.
107       split; intros; apply IHp1; apply H2. simpl. tauto.
108       split; intros; apply IHp2; apply H2. simpl. tauto.
109   (** Case p_or **)
110   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros; simpl;
111   try rewrite is_bool_expr_bool in H; try rewrite is_bool_expr_bool in H0.
112     (** p1 and p2 are simple boolean expressions. **)
113     tauto.
114     (** p1 is simple boolean expression, p2 is not. **)
115     split; intros. destruct H1.
116     rewrite (guard_or_on_bool_expr p1). left. left. apply H1. apply H0.
117     (* Hierarchy is wrong, so left and right don't work. *)
118     assert ((pred_sem (guard_and (guard_or p2)) eta eps) \/ pred_sem (guard_and (F_or p2)) eta
          eps).
119     apply guard_F2. split; intros; apply IHp2; apply H2. apply H1. tauto.
120     right. apply IHp2. apply H1.
121     (** p2 is simple boolean expression, p1 is not. **)
122     split; intros. destruct H1.
123     assert ((pred_sem (guard_and (guard_or p1)) eta eps) \/ pred_sem (guard_and (F_or p1)) eta
          eps).
124     apply guard_F2. split; intros; apply IHp1; apply H2. apply H1. tauto.
125     rewrite (guard_or_on_bool_expr p2). left. right. apply H1. apply H.
126     left. apply IHp1. apply H1.
127     (** neither are simple boolean expressions. **)
128     split; intros. destruct H1.
129     assert ((pred_sem (guard_and (guard_or p1)) eta eps) \/ pred_sem (guard_and (F_or p1)) eta
          eps).
130     apply guard_F2. split; intros; apply IHp1; apply H2. apply H1. tauto.
131     assert ((pred_sem (guard_and (guard_or p2)) eta eps) \/ pred_sem (guard_and (F_or p2)) eta
          eps).
132     apply guard_F2. split; intros; apply IHp2; apply H2. apply H1. tauto.
133     destruct H1. left. apply IHp1. apply H1. right. apply IHp2. apply H1.
134   (** Case p_forall **)
135   split; intros;
136   case_eq (is_bool_expr_b p); intros; simpl; try rewrite H0 in H; simpl in H.
137     tauto.
138     intros. apply guard_F2. apply IHp. apply H.
139     tauto.
140     destruct H. destruct (H1 x).
141     assert (guard_or (guard_or p) = guard_or p). apply guard_guard. rewrite H3 in H2. apply H2.
142     apply IHp. apply H2.
143   (** Case p_exists **)
144   split; intros;
145   case_eq (is_bool_expr_b p); intros; simpl; try rewrite H0 in H; simpl in H.
146     tauto. destruct H.
147     right. exists x. split.
148     assert (guard_and (guard_and p) = guard_and p). apply guard_guard. rewrite H1. apply H.
149     apply IHp. apply H.
150     tauto.
151     destruct H. exists x.
152     apply guard_F2. split; intros.
153     apply IHp. apply H1. apply IHp. apply H1.
154     apply H.
155 Qed.
```

Theorem 3.13

```
1 (** The results of F_and and F_or are always of their corresponding F_* form. **)
2 Lemma F_rhs_sub (p: pred_form):
3   (~is_bool_expr p) -> (is_rhs_and (F_and p) /\ is_rhs_or (F_or p)).
4 Proof.
5 intros.
6 induction p; simpl; simpl in H; try contradiction; try tauto.
7   (** Case p_and **)
8   case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros;
9   try rewrite is_bool_expr_bool in H0; try rewrite is_bool_expr_bool in H1.
10    (* p1 and p2 are simple boolean expressions *)
11    destruct H. tauto.
12    (** p1 is simple boolean expression, p2 is not. **)
13    split.
14    apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. tauto.
15    split. unfold is_guarded_conj. split. simpl. split; apply guard_bool_expr.
16    split. tauto. apply guard_F. apply H2.
17    apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. tauto.
18    (** p2 is simple boolean expression, p1 is not. **)
19    rewrite<- not_true_iff_false in H1. rewrite is_bool_expr_bool in H1.
20    split. apply IHp1. apply H1.
21    split. unfold is_guarded_conj. simpl. split. split; apply guard_bool_expr.
```

```
22      intros. split. apply H2. apply guard_F. apply H2.
23      apply IHp1. apply H1.
24      (** Neither are simple boolean expressions **)
25      rewrite<- not_true_iff_false in H0, H1. rewrite is_bool_expr_bool in H0, H1.
26      split. split. apply IHp1. apply H1. apply IHp2. apply H0.
27      split. unfold is_guarded_conj. simpl. split. split; apply guard_bool_expr.
28      split. apply H2. split; apply guard_F; apply H2.
29      split. apply IHp1. apply H1. apply IHp2. apply H0.
30    (** Case p_or **)
31    case_eq (is_bool_expr_b p1); case_eq (is_bool_expr_b p2); intros;
32    try rewrite is_bool_expr_bool in H0; try rewrite is_bool_expr_bool in H1; simpl.
33      (* p1 and p2 are simple boolean expressions *)
34      destruct H. tauto.
35      (** p1 is simple boolean expression, p2 is not. **)
36      split. split. unfold is_guarded_disj. simpl. split. split; apply guard_bool_expr.
37      intros. destruct H2. right. apply guard_F. apply H2.
38      apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. apply H0.
39      apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. apply H0.
40      (** p2 is simple boolean expression, p1 is not. **)
41      split. split. unfold is_guarded_disj. simpl. split. split; apply guard_bool_expr.
42      intros. destruct H2. apply H2. left. apply guard_F. apply H2.
43      apply IHp1. rewrite<- not_true_iff_false in H1. rewrite is_bool_expr_bool in H1. apply H1.
44      apply IHp1. rewrite<- not_true_iff_false in H1. rewrite is_bool_expr_bool in H1. apply H1.
45      (** Neither are simple boolean expressions **)
46      split. split. unfold is_guarded_disj. simpl. split. split; apply guard_bool_expr.
47      intros. destruct H2. apply H2. destruct H2. left. apply guard_F. apply H2.
48      right. apply guard_F. apply H2.
49      split.
50      apply IHp1. rewrite<- not_true_iff_false in H1. rewrite is_bool_expr_bool in H1. apply H1.
51      apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. apply H0.
52      split.
53      apply IHp1. rewrite<- not_true_iff_false in H1. rewrite is_bool_expr_bool in H1. apply H1.
54      apply IHp2. rewrite<- not_true_iff_false in H0. rewrite is_bool_expr_bool in H0. apply H0.
55    (** Case p_forall **)
56    rewrite<- is_bool_expr_bool in H. rewrite not_true_iff_false in H. rewrite H. simpl.
57    split. split. unfold is_guarded_disj. simpl. split. apply guard_bool_expr.
58    intros. destruct H0. apply H0.
59    apply guard_F. apply H0.
60    apply IHp. rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H.
61    split. unfold is_guarded_conj. simpl. split. apply guard_bool_expr.
62    split. apply H0. intros.
63    apply guard_F2. split; intros; apply guard_F; apply H1.
64    apply H0.
65    unfold is_guarded_disj. split. split.
66    apply guard_bool_expr.
67    intros. simpl in H0. destruct H0. apply H0.
68    apply guard_F. apply H0.
69    apply IHp.
70    rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H.
71    (** Case p_exists **)
72    rewrite<- is_bool_expr_bool in H. rewrite not_true_iff_false in H. rewrite H. simpl.
73    split. split. unfold is_guarded_disj. simpl. split. apply guard_bool_expr.
74    intros. destruct H0. apply H0.
75    destruct H0. exists x. apply guard_F2. split; intros; apply guard_F; apply H1.
76    apply H0.
77    split.
78    unfold is_guarded_conj. split.
79    apply guard_bool_expr.
80    intros. simpl. split. apply H0. apply guard_F. apply H0.
81    apply IHp. rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H.
82    unfold is_guarded_conj. split. split.
83    apply guard_bool_expr.
84    intros. simpl. split. apply H0. apply guard_F. apply H0.
85    apply IHp. rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H.
86  Qed.
87
88  (** The results of F are guarded and of the form RHS **)
89  Theorem F_rhs (p: pred_form):
90    is_rhs (rhs p).
91  Proof.
92  unfold is_rhs.
93  case_eq (is_bool_expr_b p); intros.
94    (** p is a simple boolean expression. **)
95    right. right. unfold rhs.
96    case (is_conjunction p).
97      simpl. split. apply guard_bool_expr. simpl.
98      induction p; simpl in H; simpl; try rewrite H; simpl; try tauto.
99      rewrite<- not_false_iff_true in H. tauto.
100     simpl. split. apply guard_bool_expr. simpl.
101     induction p; simpl in H; simpl; try rewrite H; simpl; try tauto.
```

```
102     rewrite<- not_false_iff_true in H. tauto.
103   (** p is no simple boolean expression. **)
104   unfold rhs. case (is_conjunction p).
105     right. left. simpl.
106     split. unfold is_guarded_conj. split.
107     apply guard_bool_expr.
108     intros. simpl. split. apply H0. apply guard_F. apply H0.
109     apply F_rhs_sub. rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H
        .
110     left. simpl.
111     split. unfold is_guarded_disj. split.
112     apply guard_bool_expr.
113     intros. simpl in H0. destruct H0. apply H0. apply guard_F. apply H0.
114     apply F_rhs_sub. rewrite<- not_true_iff_false in H. rewrite is_bool_expr_bool in H. apply H
        .
115 Qed.
```