Diplomarbeit

An Algorithm for Restriction of Finite Automata

Mareike Schmidt
November 2005

Betreuer:
Prof. Dr. rer. nat. Klaus Schneider
Dipl. - Inf. Tobias Schüle
Arbeitsgruppe Reaktive Systeme
Fachbereich Informatik
Universität Kaiserslautern

Erklärung

Die vorliegende Diplomarbeit wurde von mir selbständig verfasst. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Trippstadt, den 6. November 2005

Mareike Schmidt

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Extended Presburger Arithmetic XPres 5
2.2 Finite Automata 8
2.3 Translation of XPres to Finite Automata 10
2.4 State Minimization with Equivalence Classes 16

3 Restriction of Finite Automata **20**
3.1 Previous Work 20
3.2 Restrict Operator for Finite Automata 23
3.3 Algebraic Properties 28
3.4 Safeness 31
3.5 Image Computation with the Restrict Operator 32

4 The Restrict Algorithm **39**
4.1 Construction of the Product Automaton 40
4.2 Computing the Equivalence Relation 42
4.3 Counting the Number of Equivalence Classes 45
4.4 Evaluating the Counter Result 50
4.5 State Assignment 51
4.6 Minimization 51

5 Example **53**

6 Experimental Results **57**

7 Conclusion **60**

A Acknowledgement **64**

1 Introduction

In recent years, there has been a rapid growth in the development of hardware and software systems that are used in various areas of our everyday life. A majority of applications can be found in embedded systems where reliability and correctness are a necessary requirement due to the fact that these systems are often safety critical. A malfunction of programs may cause health or life danger or have at least expensive consequences. Therefore, verification of the correct functionality of systems is necessary and important.

Proving or disproving the correctness of a system with respect to a specification with formal methods is called formal verification. Model checking has evolved as the main technology for formal verification. There, the verification problem is solved by checking whether a system or program satisfies a desired property. In contrast to simulation or testing, in model checking every possible behavior of a system is explored.

In order to solve the verification problem, a given system and the desired specification have to be converted into formalisms that are accepted by model checking tools. Systems are usually represented by Kripke structures with finite or infinite state spaces. The properties to be verified are often described in temporal logics, such as linear temporal logic (LTL) or computation tree logic (CTL). For evaluation, the CTL formulas can be easily translated to the μ-calculus and LTL formulas to ω-automata.

Kripke structures can be represented symbolically by encoding the states and the transition relation with characteristic functions. For finite transition systems binary decision diagrams (BDDs) have been used in order to manipulate large sets of states efficiently. BDDs provide a canonical representation of formulas in propositional logic. A disadvantage of using propositional logic during symbolic simulation is that only finite data types can be considered. In order to express infinite data types a more powerful logic is required, e.g. Presburger arithmetic. Formulas of Presburger arithmetic can be translated to finite automata that are an efficient data structure to manipulate and store infinite sets.

Often we are only interested in certain variable assignments of a formula. Irrelevant assignments that do not belong to the care sets are called don't cares. Any other formula that has an equal evaluation of variable assignments that lie within the care set could be used for the purpose as well. For BDDs there exist two heuristics, *constrain* and *restrict,* to determine a formula that differs from the original formula only in the don't care values and can be represented by a minimal BDD. A similar operator that restricts a given automaton with respect to another that represents the don't care information would be desirable. Then, the resulting automaton should keep the specified behavior of the care set and have a minimal number of states. This would be helpful during reachability analysis because image and preimage computation could be simplified.

This work presents an exact algorithm that restricts a given automaton with respect to a restricting one. The method is safe because the resulting automaton has always less states or has at least the same size as the original one. For image computation we obtain a semi-decision procedure for the membership of a state with respect to the computed fixpoint.

In the next section, the fundamental definitions of extended Presburger arithmetic and automata are introduced. There, the translation of a formula into the corresponding finite automaton are explained and the minimization algorithm that relies on the computation of an equivalence relation is described [25]. Section 3 gives a brief overview on previous works dealing with reduction of incompletely specified finite state machines (ISFSMs). The idea of a restrict operator for finite automata is presented and the main difference between the minimization of ISFSMs and the minimization of automata with don't care states is described. Then, the algebraic properties of the restrict operator and their importance in image computation are considered. The algorithm for restriction of finite automata is proposed in Section 4 and a small example is given in Section 5. After presenting the experimental results (Section 6), the last section gives a summary and a conclusion.

2 Preliminaries

Many verification methods use an implicit set representation to manipulate large systems efficiently. For finite state systems, Binary Decision Diagrams (BDDs) have proved to be a useful data structure which provides a canonical representation of propositional formulas. Many applications in verification that use the encoding of Boolean variables benefit from the advantages of BDDs, e.g. model checking [3] or methods for state space exploration [19].

The disadvantage of BDDs is the restriction to finite data structures and the state explosion problem that occurs with large state space of the obtained data flow. A more powerful logic is Presburger arithmetic, a decidable subset of the first-order theory of natural numbers. It can be used to represent infinite data structures as well and has been used in many applications like symbolic model checking or reachability analysis of extended finite state machines [13].

2.1 Extended Presburger Arithmetic XPres

Presburger arithmetic is a decidable subset of the first order theory of natural numbers where multiplication and division are not allowed [17, 18]. The set of extended Presburger formulas XPres [21] consists of linear equations and inequations closed under the Boolean operators, bitvector operators and the quantifiers $\exists$ and $\forall$. In contrast to the original definition that uses the natural numbers $\mathbb{N}$, extended Presburger arithmetic is here interpreted over the integers $\mathbb{Z}$. Its syntax is defined as follows:

Definition 1 (Syntax of Extended Presburger Arithmetic)**.** *Let $\mathcal{V} = \mathcal{V}_{\mathbb{Z}} \cup \mathcal{V}_{\mathbb{B}}$ be a finite set of variables where $\mathcal{V}_{\mathbb{Z}}$ and $\mathcal{V}_{\mathbb{B}}$ are disjoint. $\mathcal{V}_{\mathbb{Z}}$ denotes a set of integer and $\mathcal{V}_{\mathbb{B}}$ a set of Boolean variables. Then, the set of extended Presburger terms $\mathsf{XTerm}_{\mathcal{V}}$ over a finite set of variables $\mathcal{V}$ is defined as follows:*

- $\mathbb{Z} \subseteq \mathsf{XTerm}_{\mathcal{V}}$
- $\mathcal{V}_{\mathbb{Z}} \subseteq \mathsf{XTerm}_{\mathcal{V}}$
- $\tau + \pi, \tau - \pi \in \mathsf{XTerm}_{\mathcal{V}}$ *provided that* $\tau, \pi \in \mathsf{XTerm}_{\mathcal{V}}$
- $c\tau \in \mathsf{XTerm}_{\mathcal{V}}$ *provided that* $c \in \mathbb{Z}$ *and* $\tau \in \mathsf{XTerm}_{\mathcal{V}}$
- $\widehat{\neg}\tau \in \mathsf{XTerm}_{\mathcal{V}}$ *provided that* $\tau \in \mathsf{XTerm}_{\mathcal{V}}$
- $\tau \widehat{\wedge} \pi \in \mathsf{XTerm}_{\mathcal{V}}$ *provided that* $\tau, \pi \in \mathsf{XTerm}_{\mathcal{V}}$

The set of extended Presburger formulas $\mathsf{XPres}_\mathcal{V}$ *is the smallest set satisfying the following rules:*

- $\mathcal{V}_\mathbb{B} \subseteq \mathsf{XPres}_\mathcal{V}$
- $\tau \leq \pi \in \mathsf{XPres}_\mathcal{V}$ *provided that* $\tau, \pi \in \mathsf{XTerm}_\mathcal{V}$
- $\neg\varphi \in \mathsf{XPres}_\mathcal{V}$ *provided that* $\varphi \in \mathsf{XPres}_\mathcal{V}$
- $\varphi \wedge \psi \in \mathsf{XPres}_\mathcal{V}$ *provided that* $\varphi, \psi \in \mathsf{XPres}_\mathcal{V}$
- $\exists v.\varphi \in \mathsf{XPres}_\mathcal{V}$ *provided that* $v \in \mathcal{V}$ *and* $\varphi \in \mathsf{XPres}_\mathcal{V}$

The encoding of integer values requires some notation on bitvector arithmetic to explain the semantics of XPres. Integers are given as bitstrings and are represented in two's complement encoding

$$\mathsf{enc}(b_n ... b_0) :\equiv -b_n 2^n + \sum_{i=0}^{n-1} b_i 2^i$$

with $b_i \in \{0, 1\}$. In this representation the most significant bit determines the sign of the integer. For $b_n = 1$, the value remains negative since the absolute value of the first term is greater than the sum. If $b_n = 0$, the equation yields positive numbers or zero. The encoding is not injective due to the fact that infinitely many bitstrings $\mathbf{b}$ can express the same value $\mathsf{enc}(\mathbf{b}) = c$ for any $c \in \mathbb{Z}$, e.g. $\mathsf{enc}(111) = -1 = \mathsf{enc}(11)$. This ambiguous representation is a consequence of the sign extension lemma that is valid for any bitvector $b_n ... b_0$:

$$\mathsf{enc}(b_n ... b_0) = \mathsf{enc}(b_{n+k} ... b_{n+1} b_n ... b_0)$$

with $b_n = b_{n+j}$ with $1 \leq j \leq k$. For our purpose it is sufficient that the encoding is surjective.

The semantics of Presburger arithmetic is defined by means of variable assignments which map free variables of a formula to integer values. For the interpretation of quantified formulas, it is necessary to introduce an operation that modifies variable assignments: $\varsigma[v/c](v) :\equiv c$ and $\varsigma[v/c](y) :\equiv \varsigma(y)$ for every variable y that is different from v.

Definition 2 (Semantics of Presburger Arithmetic)**.** *Given a variable assignment $\varsigma : \mathcal{V} \to \mathbb{Z}$ for a finite set of variables $\mathcal{V}$, we define its extension $\varsigma^* : \mathsf{XTerm}_{\mathcal{V}} \to \mathbb{Z}$ as follows:*

- $\varsigma^*(c) :\equiv c$ *for any* $c \in \mathbb{Z}$
- $\varsigma^*(v) :\equiv \varsigma(v)$ *for any* $v \in \mathcal{V}_{\mathbb{Z}}$
- $\varsigma^*(\tau + \pi) :\equiv \varsigma^*(\tau) + \varsigma^*(\pi)$
- $\varsigma^*(\tau - \pi) :\equiv \varsigma^*(\tau) - \varsigma^*(\pi)$
- $\varsigma^*(c\tau) :\equiv c\,\varsigma^*(\tau)$
- $\varsigma^*(\widehat{\neg}\tau) :\equiv \mathsf{enc}(\overline{b_n}...\overline{b_0})$ *for any bitvector* $b_n...b_0$ *with* $\mathsf{enc}(b_n...b_0) = \varsigma^*(\tau)$
- $\varsigma^*(\tau\widehat{\wedge}\pi) :\equiv \mathsf{enc}((a_n \wedge b_n)...(a_0 \wedge b_0))$ *for all bitvectors* $a_n...a_0$ *and* $b_n...b_0$ $with\ \mathsf{enc}(a_n...a_0) = \varsigma^*(\tau)\ and\ \mathsf{enc}(b_n...b_0) = \varsigma^*(\pi)$

Given a function $\varsigma^* : \mathsf{XTerm}_{\mathcal{V}} \to \mathbb{Z}$, we define a function $\zeta_{\varsigma^*} : \mathsf{XPres}_{\mathcal{V}} \to \mathbb{B}$ as follows:

- $\zeta_{\varsigma^*}(v) :\equiv \begin{cases} \mathsf{true} & \text{if } \varsigma(v) < 0 \\ \mathsf{false} & \text{if } \varsigma(v) \geq 0 \end{cases}$ *for any variable* $v \in \mathcal{V}^{\mathbb{B}}$
- $\zeta_{\varsigma^*}(\tau \leq \pi) :\equiv \mathsf{true}$ *iff* $\varsigma^*(\tau) \leq \varsigma^*(\pi)$
- $\zeta_{\varsigma^*}(\neg\varphi) :\equiv \begin{cases} \mathsf{true} & \text{if } \zeta_{\varsigma^*}(\varphi) = \mathsf{false} \\ \mathsf{false} & \text{if } \zeta_{\varsigma^*}(\varphi) = \mathsf{true} \end{cases}$
- $\zeta_{\varsigma^*}(\varphi \wedge \psi) :\equiv \begin{cases} \mathsf{true} & \text{if } \zeta_{\varsigma^*}(\varphi) = \mathsf{true} \text{ and } \zeta_{\varsigma^*}(\psi) = \mathsf{true} \\ \mathsf{false} & \text{otherwise} \end{cases}$
- $\zeta_{\varsigma^*}(\exists v.\varphi) :\equiv \mathsf{true}$ *iff there is a* $c \in \mathbb{Z}$ *such that* $\zeta_{\varsigma[v/c]^*}(\varphi) = \mathsf{true}$

In the following, the set of assignments that satisfy a formula $\varphi \in \mathsf{XPres}_{\mathcal{V}}$ is denoted as $[\![\varphi]\!] :\equiv \{\varsigma \mid \zeta_{\varsigma^*}(\varphi) = \mathsf{true}\}$.
The functions, predicates, and quantifiers of Presburger arithmetic are sufficient to derive other relations $\{<, >, \geq, =, \neq\}$, Boolean operations $\{\vee, \oplus, \leftrightarrow\}$ or the universal quantifier $\forall$.

2.2 Finite Automata

Similar to binary decision diagrams that are used as canonical normal forms for propositional logic, finite automata can represent formulas of Presburger arithmetic. Finite automata or finite state machines (FSMs) are models of behavior composed of states and transitions with conditions. A state reflects the input changes from the initial state to the present moment. A transition leads from a current state to a next state and obtains a condition that has to be fulfilled to enable the transition. The formal definition of a finite automaton is given below:

Definition 3 (Finite Automaton)**.**
Let $\mathcal{A}$ be an automaton with $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F})$ where

- *$\mathcal{Q}$ is the set of states,*
- *Σ is the input alphabet,*
- *$\Delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ is the transition relation,*
- *q_0 is the initial state and,*
- *$\mathcal{F} \subseteq \mathcal{Q}$ the set of final states.*

In general there are two classes of finite automata: acceptors and transducers. Acceptors change the current state according to a given input word whereas transducers additionally generate an output sequence. Predecessor states are all states of a current state that have a transition to it and its successors are the states that are reachable from this state.

Definition 4 (Predecessor and Successor States)**.**
For a given automaton $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F})$ the predecessors and the successors of a state $q \in \mathcal{Q}$ are defined as follows:

- $\mathsf{pre}_{\mathcal{A}}(q) := \{p \in \mathcal{Q} | (p, q) \in \Delta\}$
- $\mathsf{suc}_{\mathcal{A}}(q) := \{s \in \mathcal{Q} | (q, s) \in \Delta\}$

A word w is an arbitrary sequence of letters of the given alphabet. The prefix of a word is a part of the beginning of the sequence, the suffix a part of the end of the sequence.

Definition 5 (Prefix and Suffix with length i)**.** *Let $w \in \Sigma^*$ be a word of the alphabet with $|w| = n$ and $w = w_1 ... w_n$.*

The prefix of w with length $i, 0 \leq i \leq n$ is given by $\mathsf{pf}_i(w) = w_1 w_2 ... w_i$.
The suffix of w with length $i, 0 \leq i \leq n$ is given by $\mathsf{sf}_i(w) = w_{n-i+1} ... w_n$.

The automaton starts at the initial state and stops after the sequence of input symbols is fully read. The current state that has been reached after following the rules of the transition can be either a final state or not. If it belongs to the set of final states $\mathcal{F}$ the word w is said to be *accepted* otherwise to be *rejected*.

Definition 6 (Acceptance)**.** *Let $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F})$ be an automaton. The acceptance of a word $w \in (\Sigma)^*$ in a state $q \in \mathcal{Q}$ is recursively defined as follows:*

$$\mathsf{acc}_{\mathcal{A}}(q, w) := \begin{cases} q \in \mathcal{F} & \textbf{if } w = \varepsilon \\ \exists s.s \in \mathsf{suc}_{\mathcal{A}}(q) \wedge (q, (\mathsf{pf}_1(w)), s) \in \Delta & \\ \qquad \wedge\ \mathsf{acc}_{\mathcal{A}}(s, \mathsf{sf}_{n-1}(w)) & \textbf{otherwise} \end{cases}$$

The set of words over a given alphabet that are accepted by the automaton $\mathcal{A}$ is called language of $\mathcal{A}$ and is defined in the following way:

Definition 7 (Language)**.** *The language accepted by a given automaton $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, q_0, \mathcal{F})$ in a state $q \in \mathcal{Q}$ is defined as $\mathcal{L}_{\mathcal{A}}(q) := \{w \in \Sigma^* | \mathsf{acc}_{\mathcal{A}}(q, w)\}$. $\mathcal{L}_{\mathcal{A}}(q_0)$ is abbreviated by $\mathcal{L}_{\mathcal{A}}$.*

There are two special cases of accepted languages the set of all possible words $w \in \Sigma^*$ and the empty set. The automata that accept these languages are defined as follows:

Definition 8 (Special automata)**.** *Let $\top$ be the automaton that accepts all words w over the given alphabet Σ. The language of $\top$ is then $\mathcal{L}(\top) = \Sigma^*$. The automaton $\bot$ that rejects all words w accepts the language $\mathcal{L}(\bot) = \emptyset$.*

The manipulation of two automata returns an automaton that accepts a new set of words. The resulting language of manipulated automata is defined below.

Definition 9 (Languages of Manipulated Automata)**.** *Given are two automata $\mathcal{A}$ and $\mathcal{B}$ with the corresponding languages $\mathcal{L}(\mathcal{A})$ and $\mathcal{L}(\mathcal{B})$. Then the accepted languages of their negation, conjunction and disjunction are given by*

- $\mathcal{L}(\neg\mathcal{A}) = \Sigma^* \setminus \mathcal{L}(\mathcal{A})$
- $\mathcal{L}(\mathcal{A} \wedge \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$
- $\mathcal{L}(\mathcal{A} \vee \mathcal{B}) = \mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$

2.3 Translation of XPres to Finite Automata

As mentioned before, Presburger arithmetic is a decidable logic [17, 18]. A Presburger formula can be translated to a finite automaton $\mathcal{A}_\varphi$ that encodes the models $[\![\varphi]\!]$ of φ. For every finite automaton accepting a language there exists a deterministic and minimal automaton that expresses the same formula. Therefore, Presburger formulas can be represented in a canonical form as well.

An automaton reads nonempty finite words w built by a finite set of variables $\mathcal{V} = \{v_1, ..., v_m\}$ over an alphabet $\mathbb{B}^m$. The relationship between the semantics of Presburger arithmetic and finite automata is the interpretation of a word w over an alphabet $\mathbb{B}^m$ as a variable assignment $\varsigma_w(v_i) :\equiv \mathsf{enc}(w_{in}...w_{i0})$ for every variable $v_i \in \mathcal{V}$.

$$w = \begin{pmatrix} w_{1n} \\ \vdots \\ w_{mn} \end{pmatrix} \cdots \begin{pmatrix} w_{11} \\ \vdots \\ w_{m1} \end{pmatrix} \begin{pmatrix} w_{10} \\ \vdots \\ w_{m0} \end{pmatrix}$$

Reading of a word w with $w_{ij} \in \mathbb{B}$ by an automaton can be explained with the encoding scheme above. Every row represents the bitvector encoding of the value whereas the index i characterizes the variable the bit belongs to. Each processing step the automaton reads the next bit of all i variables in the same column, the index j can be seen as the number of processing steps. The number of variables is finite and fixed, whereas the bitwidth can be arbitrarily large although it is finite as well.

For example, $\mathcal{V} = \{v_1, v_2, v_3\}$ is a set of variables and the assigment ς_w is given with $\varsigma_w(v_1) = -7$, $\varsigma_w(v_2) = 4$ and $\varsigma_w(v_3) = -3$. The word w over $\mathbb{B}^3$ is given by:

$$w = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$$

Definition 10 (Representing Sets of Assignments)**.** *Let $\mathcal{V} = \{v_1, ...v_m\}$ be a finite set of assignments. Any finite automaton $\mathcal{A}$ over the alphabet $\mathbb{B}^m$ with the language $\mathcal{L}(\mathcal{A})$ represents the following set of assignments:*

$$\mathsf{Models}_{\mathcal{V}}(\mathcal{A}) :\equiv \{\varsigma : \mathcal{V} \to \mathbb{Z} | \exists w \in \mathcal{L}(\mathcal{A}) \forall v_i \in \mathcal{V}.\varsigma(v_i) = \mathsf{enc}(w_{in}..w_{i0})\}$$

The relationship between sets of assignments and finite automata leads to an interesting fact about the dependency of language equivalence and model equivalence. Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two automata with $\mathcal{L}(\mathcal{A}_1) = \mathcal{L}(\mathcal{A}_2)$, then we have $\mathsf{Models}_{\mathcal{V}}(\mathcal{A}_1) = \mathsf{Models}_{\mathcal{V}}(\mathcal{A}_2)$, but the converse does not hold.

Due to the sign extension it has to be ensured that all bitvectors that encode the same value have to be treated equally. If one of them is accepted, all the others have to be accepted as well. A closure operation on automata prevents from treating bitvectors that represent the same value differently.

Definition 11 (Sign Extension Closure)**.** *Let $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \Delta, q_0, \mathcal{F})$ be an automaton. The sign extension closure is defined as $\mathsf{cl}(\mathcal{A}) :\equiv (\mathcal{Q}, \mathbb{B}^m, \Delta', q_0, \mathcal{F})$, whereas Δ' is the least transition relation with the following properties:*

- $\Delta \subseteq \Delta'$
- *if* $(q_0, \mathbf{b}, q_1) \in \Delta$ *and* $(q_1, \mathbf{b}, q_2) \in \Delta$*, then there is* $(q_0, \mathbf{b}, q_2) \in \Delta'$
- *if* $(q_0, \mathbf{b}, q_1) \in \Delta$*, then there is* $(q_1, \mathbf{b}, q_1) \in \Delta'$

The sign extension closure can be computed by successively adding transitions to the former transition relation Δ. For the construction of the new transition relation the following lemma is important:

Lemma 1 (Sign Extension Closure)**.** *The sign extension closure has the following properties:*

- $\mathsf{cl}(\mathsf{cl}(\mathcal{A})) = \mathsf{cl}(\mathcal{A})$
- $\mathsf{Models}_{\mathcal{V}}(\mathsf{cl}(\mathcal{A})) = \mathsf{Models}_{\mathcal{V}}(\mathcal{A})$
- $\mathsf{Models}_{\mathcal{V}}(\mathcal{A}_1) = \mathsf{Models}_{\mathcal{V}}(\mathcal{A}_2)$ *iff* $\mathcal{L}(\mathsf{cl}(\mathcal{A}_1)) = \mathcal{L}(\mathsf{cl}(\mathcal{A}_2))$

Checking the equivalence of models for two automata can be reduced to the language equivalence problem over the sign extended closed automata which is decidable. Presburger Arithmetic can be decided as well. Therefore, a given formula $\varphi \in \mathsf{XPres}_{\mathcal{V}}$ is translated to an automaton $\mathcal{A}_\varphi$ such that $[\![\varphi]\!] = \mathsf{Models}_{\mathcal{V}}(\mathcal{A}_\varphi)$. The formula φ is valid iff the automaton $\mathcal{A}_\varphi$ accepts any word, and the formula is unsatisfiable iff $\mathcal{L}(\mathcal{A}_\varphi) = \emptyset$ holds.

Theorem 1 (Translating $\mathsf{XPres}_{\mathcal{V}}$ to Automata)**.** *There is an algorithm that computes for every formula $\varphi \in \mathsf{XPres}_{\mathcal{V}}$ over a given finite set of variables $\mathcal{V}$ an automaton $\mathcal{A}_\varphi$ such that $[\![\varphi]\!] = \mathsf{Models}_{\mathcal{V}}(\mathcal{A}_\varphi)$.*

A term of Presburger Arithmetic can contain arithmetic and bitvector operations at the same time. This complicates the construction of the corresponding automaton. Therefore, it is easier to separate the arithmetic parts in an expression from the bitvector parts in order to have an easier translation into automata. After the construction of the basic automata they can be combined into the resulting one that represents the whole formula.

Let $\diamond \in \{=, \neq, <, \leq, >, \geq\}$ be arithmetic predicates and $\circ \in \{\widehat{\wedge}, \widehat{\vee}, \widehat{\oplus}, \widehat{\leftrightarrow}, \widehat{\rightarrow}\}$ be bitvector operators. With the following rules an expression can be recursively splitted into a formula with equations or inequations that are pure arithmetic or bitvector type:

$$
\begin{array}{ll}
1) & (\tau_1 \pm \tau_2) \circ \tau_3 \diamond \pi :\equiv \exists u.((u = \tau_1 \pm \tau_2) \wedge (u \circ \tau_3 \diamond \pi)) \\
2) & (\tau_1 \circ \tau_2) \pm \tau_3 \diamond \pi :\equiv \exists u.((u = \tau_1 \circ \tau_2) \wedge (u \pm \tau_3 \diamond \pi))
\end{array}
$$

For example, the equation $(x + y)\widehat{\leftrightarrow}z = c$ is translated into the formula
$\exists u.((u = x + y) \wedge (u \leftrightarrow z = c))$.

In the following the construction of automata for both types are explained. The construction of automata for inequations is similar to the construction for bitvector equation even though the details are more complicated. This is the reason why only the computation of automata for equations is explained. The automata that accepts bitvector equations consists of three states. The initial state q_0 leads either to the accepting state q_a or to the sink state q_s according to the first input letter. It is introduced to prevent that the automaton accepts the empty word ε. For each input letter of the word the automaton evaluates the equation. If the current state after reading the bitvector is the accepting state the equation is satisfied, if only one letter does not satisfy the equation the automaton moves to the sink state q_s that never can be left again.

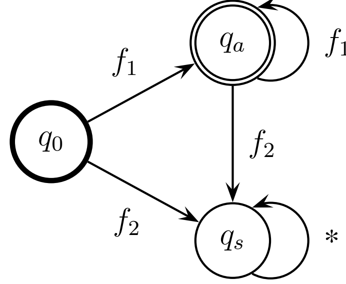

Figure 1: Automaton representing bitwise OR

Figure 1 gives an example for an automaton that represents a bitvector operation namely the bitwise disjunction which is given with the equation $z = x\widehat{\vee}y$. Let the transition relations be given by

$$f_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ * \\ 1 \end{pmatrix}, \begin{pmatrix} * \\ 1 \\ 1 \end{pmatrix} \text{ and } f_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ * \\ 0 \end{pmatrix}, \begin{pmatrix} * \\ 1 \\ 0 \end{pmatrix}$$

Automata for Boolean variables are simple to construct as well because it is sufficient to test the sign of the given bitvector. Arithmetic equations are more difficult to translate into automata. The method that now is described is adopted from [26]. The equation that has to be encoded is transformed into the $\sum_{i=1}^{m} a_i v_i = c$ with $m = ||\mathcal{V}_{\mathbb{Z}}||$, $v_i \in \mathcal{V}_{\mathbb{Z}}$ and $a_i, c \in \mathbb{Z}$. The states of the automaton correspond to the value that has been read so far from the left-side of the equation whereas the only accepting state corresponds to the result of the equation. The automaton of an arithmetic equation is given by $\mathcal{A} = (\mathcal{Q} \cup \{q_0\}, \mathbb{B}^m, \Delta, q_0, \{c\})$, with states that represent a finite set of integer values, i.e. $\mathcal{Q} \subset \mathbb{Z}$, and a special initial state q_0. The transition relation $\Delta : \mathcal{Q} \cup \{q_0\} \times \mathbb{B}^m \rightarrow \mathcal{Q}$ is defined as follows:
It is $\mathbf{a} \in \mathbb{Z}^m$, $\mathbf{b} \in \mathbb{B}^m$, and $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{m} a_i \cdot b_i$.

$$\Delta(q, b) = \begin{cases} -\mathbf{a} \cdot \mathbf{b} & \text{if } q = q_0 \\ 2q + \mathbf{a} \cdot \mathbf{b} & \text{if } q \neq q_0 \end{cases}$$

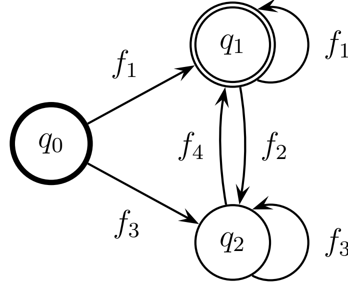

Figure 2: Automaton representing addition

The automaton reads the string from the most significant bit downto the leat significant bit, such that the transitions from the initial state read the sign of the value first. States which have no transition to the accepting state can be merged into a single sink state in order to get a complete automaton. The automaton that accepts solutions of the equation $z = x + y$ is shown in Figure 2. Its transitions are given by the following words:

$$f_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}, f_2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$f_3 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \text{ and } f_4 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$$

An implicit encoding of the functions is given by

$$\begin{aligned} f_1 &= \overline{x}\,\overline{y}\,\overline{z} \vee \overline{x}yz \vee x\overline{y}z \\ f_2 &= \overline{x}\,\overline{y}z \\ f_3 &= \overline{x}y\overline{z} \vee x\overline{y}\,\overline{z} \vee x\overline{y}\,\overline{z} \vee xyz \\ f_4 &= xy\overline{z} \end{aligned}$$

The resulting automaton of the formula can now be constructed by the combination of the basic automata according to the definitions below. Given are two automata $\mathcal{A}_\varphi = (\mathcal{Q}_\varphi, \mathbb{B}^m, \Delta_\varphi, q^0_\varphi, \mathcal{F}_\varphi)$ and $\mathcal{A}_\psi = (\mathcal{Q}_\psi, \mathbb{B}^m, \Delta_\psi, q^0_\psi, \mathcal{F}_\psi)$. Then the automata for negation, conjunction, and the existential quantification are defined by:

- $\mathcal{A}_{\neg\varphi} :\equiv (\mathcal{Q}_\varphi, \mathbb{B}^m, \Delta_\varphi, q^0_\varphi, \mathcal{Q}_\varphi \backslash \mathcal{F}_\varphi)$
- $\mathcal{A}_{\varphi\wedge\psi} :\equiv (\mathcal{Q}_\varphi \times \mathcal{Q}_\psi, \mathbb{B}^m, \Delta, (q^0_\varphi, q^0_\psi), \mathcal{F}_\varphi \times \mathcal{F}_\psi)$
  where $(p_\varphi, p_\psi, \mathbf{b}, (q_\varphi, q_\psi)) \in \Delta$ iff
  $(p_\varphi, \mathbf{b}, q_\varphi) \in \Delta_\varphi$ and $(p_\psi, \mathbf{b}, q_\psi) \in \Delta_\psi$
- $\mathcal{A}_{\exists v_i.\varphi} :\equiv \mathsf{cl}(\mathcal{Q}_\varphi, \mathbb{B}^m, \Delta, q^0_\varphi, \mathcal{F}_\varphi)$ with $(p, \mathbf{b}, q) \in \Delta$ iff
  $\exists x_i \in \mathbb{B}.(p, (b1...x_i...b_m), q) \in \Delta$

Boolean operations follow the usual definitions of automata theory, as it is described in [10]. To perform existential quantification one or more positions of the bitvector that consitute the alphabet are erased. The projection in an automaton does not preserve model consistency as it is the case with Boolean operations. That is the reason why after each projection step the closure operation has to be applied. Additionally, a method for determinization is used on the automaton that may be nondeterministic after the projection operation. This is necessary because otherwise, the computation of the complement automaton can not be computed as shown above since complementation of nondeterministic automata is a difficult operation. The complementation is used to perform universal quantification, since $\forall v.\varphi :\equiv \neg\exists v.\neg\varphi$.

2.4 State Minimization with Equivalence Classes

The number of states of an automaton can be minimized by computing an equivalence relation. It has the ability to partition the set of states $\mathcal{Q}$ into a disjoint union of subsets called equivalence classes. All states that belong to the same equivalence class can be merged into a single state. The minimization of states that relies on the computation of an equivalence relation is described in [25].

In order to minimize a deterministic finite automaton, an equivalence relation has to be computed. Its definition is given below:

Definition 12 (Equivalence Relation)**.**
Given a deterministic finite automaton $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$, the equivalence relation $E \subseteq \mathcal{Q} \times \mathcal{Q}$ is defined as: $E(q_1, q_2) \equiv \mathcal{L}_\mathcal{A}(q_1) = \mathcal{L}_\mathcal{A}(q_2)$.
The properties of equivalence relations are

- *reflexivity: for every q_1 it holds $E(q_1, q_1)$*
- *symmetry: for state pairs $E(q_1, q_2)$ it holds that $E(q_2, q_1)$*
- *transitivity: if $E(q_1, q_2)$ and $E(q_2, q_3)$ then $E(q_1, q_3)$*

A state pair belongs to the equivalence relation if they represent the same language. The equivalence of two states can be expressed with their acceptance behavior and the behavior of their successors. A state pair is equal if both states have the same acceptance behavior and there exists no transition that leads to a distinguishable pair of states.

Definition 13 (Equivalence of states)**.** *Given a deterministic finite automaton $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$, the equivalence relation E is the greatest fixpoint of the following equation:*

$$E(q_1, q_2) := (q_1 \in \mathcal{F} \equiv q_2 \in \mathcal{F}) \wedge (\forall b \in \mathbb{B}^m . \forall q_1' \in \mathsf{suc}_\mathcal{A}(q_1) . \forall q_2' \in \mathsf{suc}_\mathcal{A}(q_2) . \\ \lambda(q_1, q_1')(b) \wedge \lambda(q_2, q_2')(b) \rightarrow E(q_1', q_2'))$$

Rewriting the last term yields:

$$(\forall q_1' \in \mathsf{suc}_\mathcal{A}(q_1) . \forall q_2' \in \mathsf{suc}_\mathcal{A}(q_2) . (\exists b \in \mathbb{B}^m . \lambda(q_1, q_1')(b) \wedge \lambda(q_2, q_2')(b)) \rightarrow E(q_1', q_2'))$$

After the elimination of the quantifiers, the last term is:

$$(\bigwedge_{q'_1 \in \mathsf{suc}_{\mathcal{A}}(q_1)} \bigwedge_{q'_2 \in \mathsf{suc}_{\mathcal{A}}(q_2)} ((\lambda(q_1, q'_1) \wedge \lambda(q_2, q'_2) \not\equiv 0) \rightarrow E(q'_1, q'_2)))$$

Then the whole formula is given by:

$$\begin{aligned} E(q_1, q_2) := (q_1 \in \mathcal{F} \equiv q_2 \in \mathcal{F}) \wedge \\ (\bigwedge_{q'_1 \in \mathsf{suc}_{\mathcal{A}}(q_1)} \bigwedge_{q'_2 \in \mathsf{suc}_{\mathcal{A}}(q_2)} ((\lambda(q_1, q'_1) \wedge \lambda(q_2, q'_2) \not\equiv 0) \rightarrow E(q'_1, q'_2))) \end{aligned}$$

Any fixed point of the equivalence can be used but for minimization the greatest fixpoint has to be computed. The greatest fixpoint of E has the least number of equivalence classes of any fixpoint and can be approximated with the following computation:

Theorem 2 (Approximation of E)**.** *The greatest fixpoint can be computed iteratively by the successive approximation given as follows:*

$$\begin{aligned} E_{k+1}(q_1, q_2) := E_k(q_1, q_2) \wedge \\ (\bigwedge_{q'_1 \in \mathsf{suc}_{\mathcal{A}}(q_1)} \bigwedge_{q'_2 \in \mathsf{suc}_{\mathcal{A}}(q_2)} (\lambda(q_1, q'_1) \wedge \lambda(q_2, q'_2) \not\equiv 0) \rightarrow E_k(q'_1, q'_2)) \end{aligned}$$

where E_0 is defined by: $E_0(q_1, q_2) := (q_1 \in \mathcal{F} \equiv q_2 \in \mathcal{F})$

E_0 can also be defined as $E_0 = (\mathcal{Q} \backslash \mathcal{F})^2 \cup \mathcal{F}^2$. The approximation is computed by a greatest fixpoint that means for all $k \geq 0$ holds $E_{k+1} \subseteq E_k$. Since the initial set is an equivalence relation and the following sets are subsets the properties of equivalence relations are not violated.
State pairs that belong to the equivalence relation E_k are said to be *k-equivalent*. This means that these pairs have the same equivalence behavior for all prefixes with a maximal length k.

Another possibility to compute the equivalence relation E is to approximate a distinguishability relation D. The problems are dual since D is the complement of the equivalence relation $D = \neg E$. Then the definition of D is the negation of E and the approximation is computed by the least fixpoint that leads exactly to the same result regarding the equivalence classes. In every step where a state pair is removed from the set E_k, it is added to the set D_k.

Equivalence classes $e \in [\mathcal{Q}]_E$ are the result of the equivalence relation that splits the set of states $\mathcal{Q}$ into disjoint partitions. All states of $\mathcal{Q}$ belong to one and only one partition. States with equal behavior belong to the same partition or from the view of the distinguishability relation: distinguishable states do not belong to the same equivalence class. The finest partition of $\mathcal{Q}$ is obtained when the relation is given by the identity relation $I_{\mathcal{Q}}$ that maps every state into a single equivalence class. This trivial partition is useless for minimization since we desire to reduce the number of states in the automaton by merging all states of an equivalence class into a single one.

The distribution of all states into different equivalence classes follows directly from the definition of the equivalence relation. States belong to the same equivalence class when they have the same acceptance behavior and for every word the corresponding successor states are in the same partition. We can say as well, a partition is splittable when there exist states that read the same input and lead to successor states in different equivalence classes.

For any equivalence relation E on a set of states P the set of equivalence classes of E is denoted as $[P]_E$ with $[P]_E = \{[p]_E : p \in P\}$. The set $[P]_E$ is also called the partition of P induced by E.

The minimized automaton $\mathcal{A}_{min} = ([\mathcal{Q}]_E, \mathbb{B}^m, \Delta', [q_0]_E, [\mathcal{F}]_E)$ is the result after merging the states of the partitions into single states. The former transition relation Δ changes into $\Delta' = \{([p]_E, b, [q]_E) | (p, b, q) \in \Delta\}$.

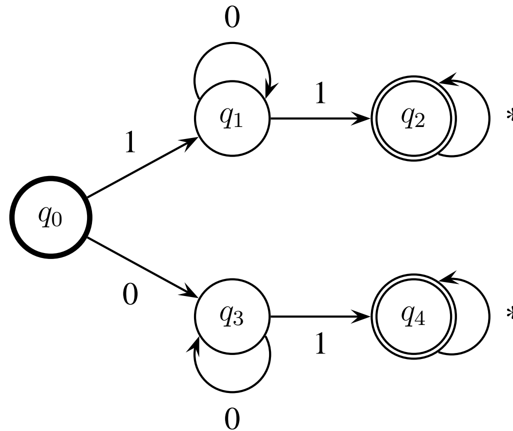

Figure 3: Example

An example ist given in Figure 3. It can be seen easily that the automaton consists of three equivalence classes. The initial state q_0 is distinguishable to every state of the automaton due to the fact that empty words $w = \varepsilon$ shall not be accepted. The equivalence classes are given by the following partitions $[\mathcal{Q}]_E = \{e_0, e_1, e_2\}$ with $e_0 = \{q_0\}, e_1 = \{q_1, q_3\}$ and $e_2 = \{q_2, q_4\}$. Therefore the automaton can be minimized in an equivalent one by merging the states within each equivalence class which is shown in Figure 4.

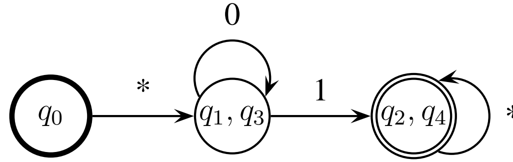

Figure 4: Example after merging states

An upperbound on number of approximation steps in the computation of the equivalent relation E is given by $(|\mathcal{Q}| - 2)$. The sequence of approximations is given by $E_0 \supset E_1 \supset ... \supset E_g \supseteq I_{\mathcal{Q}}$ where E_g is the greatest fixpoint and $I_{\mathcal{Q}}$ the identity relation on states. For the number of partitions it holds that $|[P]_{E_0}| < |[P]_{E_1}| < ... < |[P]_{E_g}| \leq |[P]_{I_{\mathcal{Q}}}|$. The identity relation maps every state in an own equivalence class such that $|[I_{\mathcal{Q}}]| = |\mathcal{Q}|$. For the number of partitions in the initial set holds $|[P]_{E_0}| \leq 2$.
We have to distinguish between three cases:

- $|[P]_{E_0}| = 0$: E_0 is the greatest fixpoint.
- $|[P]_{E_0}| = 1$: Either all states are final states or non-final states. In both cases E_0 is the greatest fixpoint.
- $|[P]_{E_0}| = 2$: For all following approximations holds that $|[P]_{E_i}| \geq i + 2$ since in each approximation step at least one new equivalence class is build. Then for the greatest fixpoint holds $g + 2 \leq |[P]_{E_g}| \leq |[P]_{I_{\mathcal{Q}}}| = |\mathcal{Q}|$ and we get $g \leq |\mathcal{Q}| - 2$.

The computation of a greatest fixpoint starting at E_0 can be done with maximal $(|\mathcal{Q}| - 2)$ iteration steps. This upperbound also holds for the computation of an distinguishability relation D and the approximation of the equivalence classes $[\mathcal{Q}]_E$.

3 Restriction of Finite Automata

Finite state machines have been widely used in modeling of application behavior, design of hardware digital systems, software engineering, compilers, study of computation and languages, and many more. That is a reason why state reduction has always been an important problem that was studied by many researchers. Incompletely specified finite state machines (ISFSMs) have at least one state where either the next state or the output is not specified.

In the next subsection, the previous work in literature dealing with reduction of ISFSMs is summarized. Then, the idea of an restrict operator for finite automata is proposed and the main differences between both minimization problems are explained. The properties and the practical use of the restrict operator are described later.

3.1 Previous Work

For completely specified FSMs there is a technique that minimizes automata by determining equivalent states in polynomial time [9]. The problem of reducing incompletely specified finite state machines is known to be NP-complete. Pfleeger showed in [16] that the minimization problem is polynomial reducible to the graph coloring problem that is known to be NP-complete.

Most of the presented algorithms that reduce incompletely specified state machines consist of two steps. The generation of compatible state sets and the selection of a minimal closed cover.
In the first step all states that generate non-conflicting outputs can be mapped together into the same compatible set. The solution of the minimizing problem is the selection of a minimal number of compatible sets such that all states of the ISFSM are covered and closure constraints are kept.

Paull and Unger [14] developed a general theory and proposed methods that generate maximal compatibles and techniques to obtain the minimal closed cover. Grasselli and Luccio improved the method to reduce the number of states. In [6] they show that only some compatibility classes (prime compatibles) have to be considered for the solution. Their proposed minimization algorithm is applicable on every type of incompletely specified flow table. The minimal covering problem is represented by a covering and closure table. A cyclic table is obtained by

eliminating rows and columns after applying reduction rules. The reduced table corresponds to a integer linear program with a minimal number of constraints that solves the problem of minimal covering.

Rho *et al.* [11] presented several algorithms for state reduction of ISFSMs that are implemented in a program called STAMINA which runs in exact and heuristic modes. The exact mode uses an approach that is based on Luccios and Grassellis binate covering problem. It is feasible for most hand-designed FSMs. For large machine-generated automata two heuristics are introduced in order to reduce time and memory requirements. Both options use explicit enumeration of compatible classes.

An implicit enumeration technique was proposed by Kam and Villa [24]. Their exact algorithm benefits from the efficiency of BDDs and is able to solve the problem for large sets of compatibles and prime compatibles. The generation of prime compatibles is formulated implicitly as well as the solution of the binate covering problem where the rows and columns of the corresponding table are encoded by BDDs. The implementation of their method is called ISM.

Higutchi and Matsunaga presented the heuristic program SLIM that represents large numbers of compatibles by BDDs [8]. The algorithm generates the set of all maximal compatibles as initial solution and reduces them iteratively to a subset by keeping the closure constraints. The compatibles are represented implicitly when the number of compatibles is too large. The technique based on iterative improvement of the initial set of compatibles indicates efficiency in computational time in experimental results.

Two heuristics were proposed by Kannan and Sarma in [12]. After the generation of the maximal compatibles, an efficient method is introduced to compute a minimal cover. Then this solution has to be expanded to obtain an optimal closed cover. The large set approach adds appropriate compatible sets to the minimal cover until the set is closed.The lean set approach starts with the sets of maximal compatibles in the minimal cover where all states with multiple occurrence are deleted. Then states or compatibles are tried and added in order to satisfy the closure condition.

An exact reduction algorithm that is not based on enumeration of compatible sets was proposed by Pena and Oliviera [15]. The advantage is that their method is independent of the number of compatibles. The authors use a string generation strategy that was inspired by the L^* algorithm of Angluin [1] that produces a set of I/O mappings (input/output pairs) that specify the ISFSM behavior. With this set a corresponding loop-free FSM is build which is used to search a minimal FSM. The computation of an equivalent FSM with a given number of states is performed in a search tree where the authors make use of Biermann's search algorithm [2].

Gören and Ferguson [7] present a heuristic algorithm for state reduction in ISFSMs that is based on checking sequence generation and identification of compatible state sets. The method is able to reduce ISFSMs that have unreachable states and deadlocks. Their algorithm constructs a new FSM that has less states than the FSM that shall be minimized. Then for each automaton input sequences are build that find contradictions within both automata. If no sequence can be found the automata are equal.

3.2 Restrict Operator for Finite Automata

Often we are only interested in special evaluations of a formula φ whereas the values caused by the other variable assignments could be chosen arbitrarily. These irrelevant variable assignments that do not belong to the care set are called don't cares. A formula β can be used to characterize the care set of φ. It is evaluated to true when the value of φ is relevant and false if it is don't care.

Instead of the formula φ any other formula ψ that only differs from φ on the don't care values can be considered. The existence of don't care values can be used to determine a function ψ such that the relevant variable assignments of φ have the same evaluation as in ψ and the data structure that represents the formula can be minimized.

In our case Presburger formulas are represented with finite automata. Words that are accepted encode the models $[\![\varphi]\!]$ of φ. Let χ_φ be a characteristic function that describes the acceptance of the automaton $\mathcal{A}_\varphi$ such that

$$\chi_\varphi(w) = \begin{cases} \mathsf{true} & \text{if } w \in \mathcal{L}(\mathcal{A}_\varphi) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

A small example is given below. Given is a formula $\varphi = (x > 0)$. Then, the words that are accepted by $\mathcal{A}_\varphi$ are given by $\mathcal{L}(\mathcal{A}_\varphi) = \{w | w = 0(0^*1^*)^*\}$. In the notation of a character sequence characters that may appear n times with $n \in \mathbb{N}, n \geq 0$ are written with w_i^* and w_i^+ when $n \in \mathbb{N}, n > 0$. Let χ_φ be the characteristic function that represents the acceptance of $\mathcal{A}_\varphi$ and χ_β the characteristic function of the automaton that specifies the care set. χ'_φ is the function with relevant values and don't cares (dc). The language of $\mathcal{A}_\beta$ is in our case $\mathcal{L}(\mathcal{A}_\beta) = \{w | w = 0(0^*1^*)^* \vee 1(0^*1^*)^*\}$ with the corresponding Presburger formula $\beta = (x \neq 0)$. The formula $\psi = (x \geq 0)$ has the same values as φ for variable assignments of the care set. In Section 5, it is shown that the automaton $\mathcal{A}_\psi$ is really smaller than $\mathcal{A}_\varphi$.

x	w	χ_φ	χ_β	χ'_ϕ	χ_ψ
$(x = 0)$	0^+	0	0	dc	1
$(x < 0)$	$1(0^*1^*)^*$	0	1	0	0
$(x > 0)$	$0(0^*1^*)^*1(0^*1^*)^*$	1	1	1	1

For binary decision diagrams there exist two heuristics that change the original function under the constraints of β into a new formula that the resulting BDD has minimal size [22]. Our goal is to find a similar operator that restricts an existing automaton $\mathcal{A}_1$ with respect to a restricting one $\mathcal{A}_2$ such that the result has a minimal number of states.

Definition 14 (Restricted Automaton)**.**
Given an automaton $\mathcal{A}_1$ and a restricting automaton $\mathcal{A}_2$, an automaton $\mathcal{A}'$ is called restriction of $\mathcal{A}_1$ w.r.t. $\mathcal{A}_2$ iff $w \in \mathcal{L}_{\mathcal{A}_2} \Rightarrow (w \in \mathcal{L}_{\mathcal{A}'} \Leftrightarrow w \in \mathcal{L}_{\mathcal{A}_1})$ holds for all $w \in (\mathbb{B}^m)^*$.

Note that $\mathcal{A}'$ is not uniquely defined.
The restrict operator that is used on two automata is written as $\mathcal{A}' := \mathcal{A}_1 \downarrow \mathcal{A}_2$ and the language that is obtained has the following notation $\mathcal{L}(\mathcal{A}') := \mathcal{L}(\mathcal{A}_1 \downarrow \mathcal{A}_2)$.

Definition 15 (Size of an automaton)**.**
The size of an automaton $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q^0, \mathcal{F})$ is defined by the number of its states $|\mathcal{A}| = |\mathcal{Q}|$.

After minimization the restricted automaton $\mathcal{A}'$ is desired to be smaller or equal than $\mathcal{A}_1$: $|\mathcal{A}'| \leq |\mathcal{A}_1|$, which means that it has a less or equal number of states $|Q'| \leq |Q_1|$.

The implementation of the restrict operator uses a semi-symbolic representation of automata in order manipulate them efficiently.

Definition 16 (Semi-Symbolic Finite Automaton)**.** *A semi-symbolic finite automaton is represented as a tuple $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$ where*

- *$\mathcal{Q}$ is the set of states*
- *$\mathbb{B}^n$ is the alphabet*
- *$\delta \subseteq \mathcal{Q} \times \mathcal{Q}$ is the transition relation*
- *$\lambda : \delta \to (\mathbb{B}^m \to \mathbb{B})$ is a labeling function*
- *$q_0 \in \mathcal{Q}$ is the initial state, and*
- *$\mathcal{F} \subseteq \mathcal{Q}$ is the set of final states*

The automaton is called semi-symbolic because its states are represented explicitly, whereas the transitions between two states are given symbolically and can be encoded by BDDs. A labeling function λ which is given as a propositional function, associates with each transition a set of letters of the given alphabet.

Automata that are used for our purpose are deterministic and complete. The result after restriction has the same properties and is minimal. Languages that can be represented by finite automata are regular and therefore, the sets of assignments satisfy XPres formulas that are regular. However, the restriction to regular sets of assignments is not a limitation since all computable functions can be encoded by a regular relation [21].

Definition 17 (Completeness)**.** *An automaton* $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$ *is complete iff* $\bigvee_{s \in \mathsf{suc}_{\mathcal{A}}(q)} \lambda(q, s)$ *holds for all* $q \in \mathcal{Q}$*.*

Completeness ensures that all possible inputs of the alphabet have a transition to a successor state. An incomplete automaton can be made complete by adding the missing transitions to each state and let them lead into a sink state.

Definition 18 (Determinism)**.** *An automaton* $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$ *is deterministic iff for all* $q \in \mathcal{Q}$ *and for all* $s_1, s_2 \in \mathsf{suc}_{\mathcal{A}}(q)$ *with* $s_1 \neq s_2$ *it holds that* $\lambda(q, s_1) \wedge \lambda(q, s_2)$ *is unsatisfiable.*

Determinism of an automaton means that for an input in a current state there is only one possible transition to the next state.

Definition 19 (Minimal Deterministic Automaton)**.** *A deterministic automaton* $\mathcal{A} = (\mathcal{Q}, \mathbb{B}^m, \delta, \lambda, q_0, \mathcal{F})$ *is minimal iff for all* $q_1, q_2 \in \mathcal{Q}$ *with* $q_1 \neq q_2$ *it holds that* $\mathcal{L}_{\mathcal{A}}(q_1) \neq \mathcal{L}_{\mathcal{A}}(q_2)$*.*

The minimal size is important to obtain a canonical normal form for Presburger formulas. As it was mentioned before for each automaton with a certain language there exists a minimal equivalent automaton.

The minimization of the given automaton $\mathcal{A}_1$ w.r.t. the restricting automaton $\mathcal{A}_2$ requires the computation of their product automaton. A product automaton corresponds to the parallel processing in both automata when reading the same input word w. The cartesian product of the set of states $\mathcal{Q}_1$ and $\mathcal{Q}_2$ compose the new states of the product automaton. The nonempty intersection of the set of symbols

of transitions corresponding to a pair of states is a transition of the product automaton that leads to the state that is composed by the successor states in $\mathcal{A}_1$ and $\mathcal{A}_2$. Usually not every state that is computed by the cartesian product of both state sets is reachable. Depending on the chosen set of accepting states the product automaton represents different kinds of languages, for example $\mathcal{F}_P = \mathcal{F}_1 \times \mathcal{F}_2$ accepts the intersection of both languages $\mathcal{L}(\mathcal{A}_P) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ whereas $\mathcal{F}_P = (\mathcal{F}_1 \times \mathcal{Q}_2) \cup (\mathcal{Q}_1 \times \mathcal{F}_2)$ accepts the union $\mathcal{L}(\mathcal{A}_P) = \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$ of both languages. Product automata are very useful for language recognition.

Definition 20 (Product Automaton)**.** *Given are two automata* $\mathcal{A}_1 = (\mathcal{Q}_1, \mathbb{B}^m, \delta_1, \lambda_1, q_1^0, \mathcal{F}_1)$ *and* $\mathcal{A}_2 = (\mathcal{Q}_2, \mathbb{B}^m, \delta_2, \lambda_2, q_2^0, \mathcal{F}_2)$*, the product automaton* $\mathcal{A}_P = \mathcal{A}_1 \times \mathcal{A}_2$ *is represented as a tuple* $\mathcal{A}_P = (\mathcal{Q}_P, \mathbb{B}^m, \delta_P, \lambda_P, q_P^0, \mathcal{F}_P)$ *where*

- *the set of states is* $\mathcal{Q}_P = \mathcal{Q}_1 \times \mathcal{Q}_2$
- *the alphabet* $\mathbb{B}^n$ *remains*
- *the transition relation* δ *is given by* $((q_1, q_2), (q_1', q_2')) \in \delta_P$ *iff* $(q_1, q_1') \in \delta_1 \wedge (q_2, q_2') \in \delta_2$ *and* $\lambda_1(q_1, q_1') \wedge \lambda_2(q_2, q_2') \not\equiv 0$
- *the labeling function* λ *is given by* $\lambda_P((q_1, q_2), (q_1', q_2')) = \lambda_1(q_1, q_1') \wedge \lambda_2(q_2, q_2')$
- *the initial state* $q_P^0 = (q_1^0, q_2^0) \in \mathcal{Q}_P$ *is computed by* $q_1^0 \in \mathcal{Q}_1$ *and* $q_2^0 \in \mathcal{Q}_2$*, and*
- *the set of final states* $\mathcal{F}_P \subseteq \mathcal{Q}_P$ *is computed by* $\mathcal{F}_1 \times \mathcal{F}_2 \subseteq \mathcal{Q}_1 \times \mathcal{Q}_2$*.*

The resulting product automaton has three classes of state sets. States of $\mathcal{A}_1$ combined with final states of the restricting automaton keep their acceptance behavior and belong to the set of final or non-final states.

States that are composed with non-final states of the restricting automaton have an undefined acceptance behavior and belong temporarily to the set of don't care states $DC := \{(q_i, q_j) | q_1 \in \mathcal{Q}_1 \wedge q_j \in \mathcal{Q}_2 \backslash \mathcal{F}_2\}$. States of this set $DC \subset \mathcal{Q}_P$ have to be assigned to the set of final states or non-final states in order to reduce the number of states to a minimum.

There is a main difference between minimizing a product automaton with don't care states and the reduction of ISFSMs. Our automata are acceptors and have no output function and in contrast to ISFSMs, the next states of the product automaton are specified. Here, the degree of freedom is the acceptance behavior of don't care states.

In the last section the minimizing algorithm with equivalence classes was proposed. We use it in order to minimize the product automaton. Since not all states have a specified acceptance behavior the result of the computation is not complete. Beside the values of the equivalence relation, formulas are obtained that represent the equivalence of state pairs. This interim result is used to determine the acceptance behavior of every don't care state to compute a minimal number of equivalence classes. Since the number of equivalence classes determines the number of states the solution is exact and minimal. After assigning the don't care states into the set of final or the set of non-final states, the evaluation of the equivalence relation can be completed and the minimized automaton can be constructed.

The problem of computing the restriction of an automaton $\mathcal{A}_1$ with respect to an automaton that represents the don't care information lies in the class of NP-complete problems.

Theorem 3 (Complexity of the Restriction)**.** *Given two automata $\mathcal{A}_1$ and $\mathcal{A}_2$, computing the minimal restricted automaton of $\mathcal{A}_1$ with respect to $\mathcal{A}_2$ is NP-complete.*

Proof. The proof follows immediately from problem [AL8] (Minimum Inferred Finite State Automaton) in [4]. There, a finite alphabet Σ, two finite subsets $S, T \subseteq \Sigma^*$ and a positive integer K are given. The question is if there exists a K-state deterministic finite automaton A that recognizes a language $L \subseteq \Sigma^*$ such that $S \subseteq L$ and $T \subseteq \Sigma^* \backslash L$ hold. For our purpose, we set $S := \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ and $T := \mathcal{L}(\mathcal{A}_2) \backslash \mathcal{L}(\mathcal{A}_1)$ to obtain the corresponding problem. [AL8] is known to be NP-complete since the problem can be transformed to the monotone 3SAT problem [5].

□

The restrict operator is used in order to minimize the size of a given automaton with respect to another automaton that specifies a desired behavior. We are interested in the algebraic properties of an operator since they are important for the practical use of the operator.

3.3 Algebraic Properties

An identity element with respect to a binary operation maps all the elements of a given set to themselves. For the restrict operator used on automata we search the automaton that maps the given automata to itself such that the accepted language is the same as before.

Theorem 4 (Identity Element)**.**
The identity element with respect to the restrict operator on automata is the automaton I that fulfills the property $\mathcal{L}(\mathcal{A} \downarrow I) = \mathcal{L}(\mathcal{A})$.
The automaton I is the automaton $\top$ that accepts the language $\mathcal{L}(\top) = \Sigma^*$.

Proof. Using the definition of restrict it holds that:
$1 \rightarrow (\mathcal{A} \leftrightarrow \mathcal{A}) = 1 \rightarrow 1$

□

The inverse element with respect to a binary operation is the element that maps each element to the identity element. In our case the solution are automata that restrict the given automata to $\top$ with the language $\mathcal{L}(\top) = \Sigma^*$.

Theorem 5 (Inverse Element)**.**
The inverse element with respect to the restrict operator of an automaton $\mathcal{A}$ is an automaton $\tilde{\mathcal{A}}$ that fulfills $\mathcal{L}(\mathcal{A} \downarrow \tilde{\mathcal{A}}) = \mathcal{L}(I)$.
For exact algorithms the inverse element is every automaton $\tilde{\mathcal{A}}$ that accepts a language $\mathcal{L}(\tilde{\mathcal{A}})$ with $\mathcal{L}(\tilde{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$.

Proof. The automaton $\tilde{\mathcal{A}}$ restricted with $\mathcal{A}$ shall have the resulting automaton $\top$ that accepts the language Σ^*. With the definition we get the following result:
$\tilde{\mathcal{A}} \rightarrow (\mathcal{A} \leftrightarrow 1) = \tilde{\mathcal{A}} \rightarrow \mathcal{A} \qquad \Rightarrow \mathcal{L}(\tilde{\mathcal{A}}) \subseteq \mathcal{L}(\mathcal{A})$. □

Theorem 6 (Interval of the Restriction)**.** *For automata $\mathcal{A}$ restricted with respect to $\mathcal{B}$ the language $\mathcal{L}(\mathcal{A} \downarrow \mathcal{B})$ lies in the interval*

$$\mathcal{L}(\mathcal{A} \wedge \mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \downarrow \mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \vee \neg\mathcal{B})$$

Proof.
First it is shown that $\mathcal{L}(\mathcal{A} \wedge \mathcal{B}) = \mathcal{L}(\mathcal{A} \downarrow \mathcal{B})$ is a valid solution:

$$\begin{aligned} \mathcal{B} \to (\mathcal{A} \leftrightarrow \mathcal{A}') &= \mathcal{B} \to (\mathcal{A} \leftrightarrow (\mathcal{A} \wedge \mathcal{B})) \\ &= \neg\mathcal{B} \vee (\mathcal{A} \wedge (\mathcal{A} \wedge \mathcal{B}) \vee \neg\mathcal{A} \wedge \neg(\mathcal{A} \wedge \mathcal{B})) \\ &= \neg\mathcal{B} \vee (\mathcal{A} \wedge \mathcal{B}) \vee (\neg\mathcal{A} \wedge (\neg\mathcal{A} \vee \neg\mathcal{B})) \\ &= \neg\mathcal{B} \vee (\mathcal{A} \wedge \mathcal{B}) \vee \neg\mathcal{A} \vee (\neg\mathcal{A} \wedge \neg\mathcal{B}) \\ &= 1 \end{aligned}$$

The definition of restrict requires that all words that are in the language of the restricting automaton $\mathcal{L}(\mathcal{B})$ keep their specified behavior of $\mathcal{A}$. The smallest set of accepted words is the intersection of both languages (Figure 5, Example b)). The upper boundary given by $\mathcal{L}(\mathcal{A} \vee \neg\mathcal{B})$ is a valid result as well:

$$\begin{aligned} \mathcal{B} \to (\mathcal{A} \leftrightarrow \mathcal{A}') &= \mathcal{B} \to (\mathcal{A} \leftrightarrow (\mathcal{A} \vee \neg\mathcal{B})) \\ &= \neg\mathcal{B} \vee (\mathcal{A} \wedge (\mathcal{A} \vee \neg\mathcal{B}) \vee \neg\mathcal{A} \wedge \neg(\mathcal{A} \vee \neg\mathcal{B})) \\ &= \neg\mathcal{B} \vee (\mathcal{A} \vee \neg\mathcal{A} \wedge (\neg\mathcal{A} \vee \mathcal{B})) \\ &= \neg\mathcal{B} \vee \mathcal{A} \vee \neg\mathcal{A} \vee (\neg\mathcal{A} \wedge \mathcal{B}) \\ &= 1 \end{aligned}$$

The largest set of words that can be accepted is given by the smallest set of accepted words and all words where the acceptance is unspecified (Figure 5, Example d)). ☐

Unfortunately, the size of an automaton $\mathcal{A}$ accepting the language $\mathcal{L}(\mathcal{A})$ does not depend on the size of the set that represents the accepted words. Therefore, it can not be said that a large set of accepted words is expressed by a small automaton or vice versa.

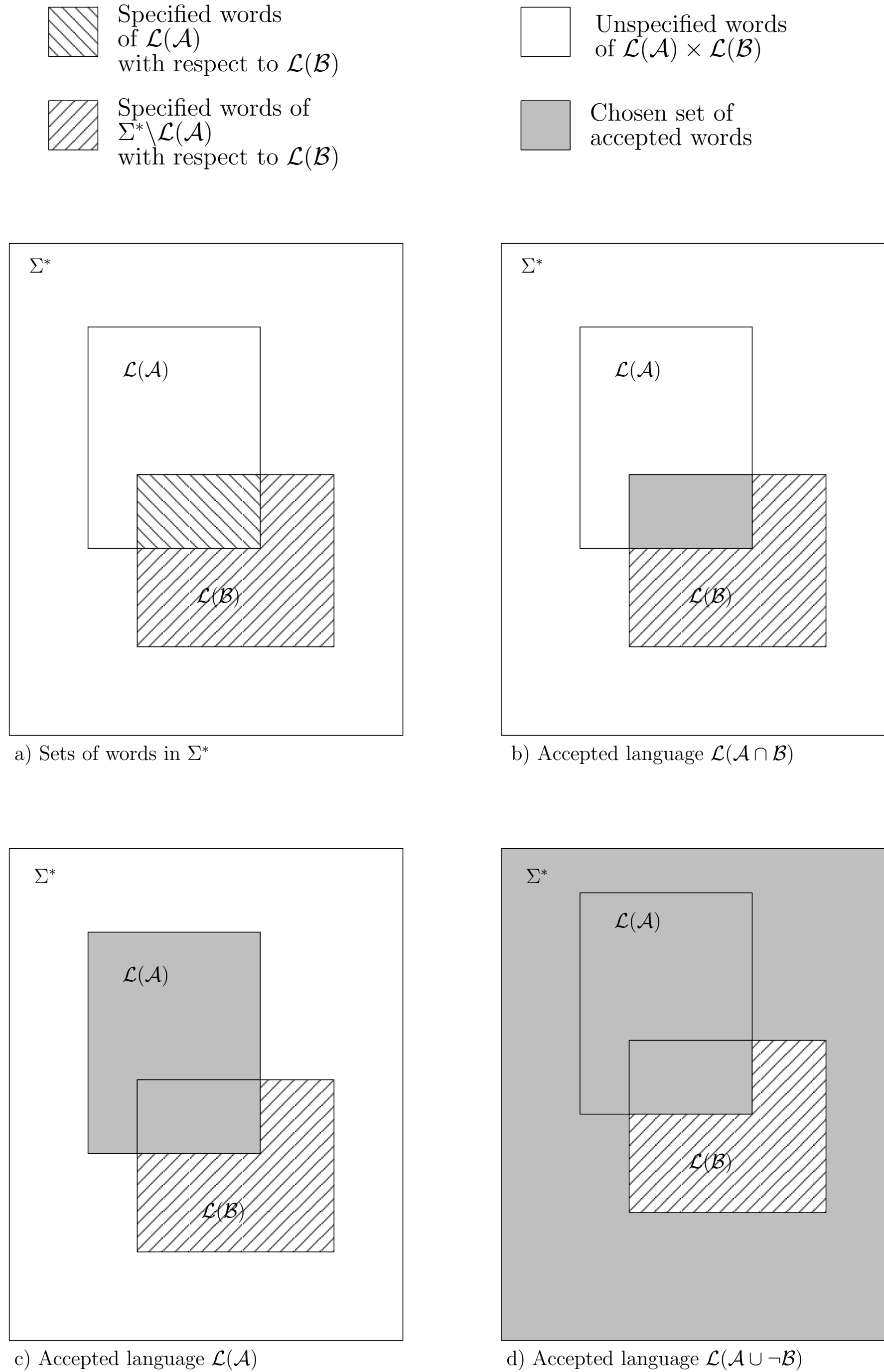

Figure 5: Possible languages of $\mathcal{A}'$

From Theorem 6, we can derive immediately that the following relations hold for two given sets:

$$
\begin{aligned}
&\mathcal{L}(\mathcal{A} \downarrow \mathcal{A}): \\
&(\mathcal{A} \rightarrow (\mathcal{A} \leftrightarrow \mathcal{A}')) \quad && = \neg\mathcal{A} \vee (\mathcal{A} \wedge \mathcal{A}' \vee \neg\mathcal{A} \wedge \neg\mathcal{A}') \\
& && = \neg\mathcal{A} \vee \mathcal{A} \wedge \mathcal{A}' \\
& && = \neg\mathcal{A} \vee \mathcal{A}' \\
& && = \mathcal{A} \rightarrow \mathcal{A}' \\
& && \Rightarrow \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A} \downarrow \mathcal{A})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{L}(\top \downarrow \mathcal{B}): \\
&(\mathcal{B} \rightarrow (1 \leftrightarrow \mathcal{A}')) = \quad && \mathcal{B} \rightarrow \mathcal{A}' \\
& && \Rightarrow \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\top \downarrow \mathcal{B})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{L}(\bot\downarrow \mathcal{B}): \\
&(\mathcal{B} \rightarrow (0 \leftrightarrow \mathcal{A}')) = \quad && \mathcal{B} \rightarrow \neg\mathcal{A}' \\
& && \Rightarrow \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\neg(\bot\downarrow \mathcal{B}))
\end{aligned}
$$

The restrict operator is neither associative nor commutative. Distributivity over conjunctions or disjunctions does not hold, due to the fact that the restricted automaton is not uniquely defined.

3.4 Safeness

One important property is safeness that guarantees that the result of the operation is always smaller or at least has equal size than the original automaton independent of the choice of the restricting automaton. Safeness is a property of the algorithm, so it has to be proved that the exact algorithm is by definition always safe.

Theorem 7 (Safeness)**.** *The restrict operator is safe iff the size of the resulting automaton is less than or equal to the size of the automaton that has to be restricted, i.e.,* $|\mathcal{A} \downarrow \mathcal{B}| \leq |\mathcal{A}|$

Proof. Suppose that there exists an automaton $\mathcal{B}$, so that the resulting automaton of the restriction $\mathcal{A}' = \mathcal{A} \downarrow \mathcal{B}$ has more states than the automaton $\mathcal{A}$ to be restricted ($|\mathcal{A}'| > |\mathcal{A}|$). Since the algorithm computes an exact solution, the resulting automaton is always minimal. Using the fact that every language in the

interval $\mathcal{L}(\mathcal{A} \wedge \mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \downarrow \mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \vee \neg\mathcal{B})$ is a language that obeys the definition, we can choose $\mathcal{A}' = \mathcal{A}$, since $\mathcal{A}$ lies within this interval (Figure 5, Example c)). This solution is independent from the restricting automaton $\mathcal{B}$ and keeps the size, since the solution remains the original automaton $\mathcal{A}$. Due to the fact that the algorithm is exact, the size of the solution may be even smaller than $\mathcal{A}$ because automata with less states are found if they exist. This contradicts the assumption that the restricted automaton has a larger size than $\mathcal{A}$. □

3.5 Image Computation with the Restrict Operator

Symbolic image computation is a fundamental operation used in reachability analysis and formal verification. The computation of the preimage is also important, for example in CTL model checking. Image computation consists of finding all successor states of a given set of states $\mathcal{S}$ according to a set of transitions $\mathcal{R}$, in preimage computation all predecessor states are searched. Since the computations are fundamental operations, it would be helpful to make use of formulas that are represented by restricted automata instead of using the common operation.

In verification, systems are often represented in Kripke structures. A Kripke structure is a transition system where every state describes a system configuration of the model. A transition between two states defines a valid change of variables during the execution of the program. Since Presburger arithmetic is the base logic the corresponding transition system is an integer Kripke structure.

Definition 21 (Integer Kripke Structure)**.**
Given a finite set of variables $\mathcal{V} = \mathcal{V}^{\mathbb{Z}} \cup \mathcal{V}^{\mathbb{B}}$, an integer Kripke structure (IKS) is a transition system $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ where $\mathcal{S}$ is the possibly infinite set of states, $\mathcal{I} \subseteq \mathcal{S}$ are the initial states, and $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ is the transition relation. Every state $s \in \mathcal{S}$ is a variable assignment $s : \mathcal{V}^{\mathbb{Z}} \cup \mathcal{V}^{\mathbb{B}} \to \mathbb{Z} \cup \mathbb{B}$, such that $s(x) \in \mathbb{Z}$ for $x \in \mathcal{V}^{\mathbb{Z}}$ and $s(x) \in \mathbb{B}$ for $x \in \mathcal{V}^{\mathbb{B}}$. In addition, it is required that the set of initial states $\mathcal{I}$ and the transition relation $\mathcal{R}$ are definable in Presburger arithmetic.

A state is defined above as a variable assignment for the variables $\mathcal{V}$. For the modeling of a system the assignment describes the current values of the variables of the system. When the system proceeds with the execution some of the variables may change such that we have a new assignment at the next point of time. A transition relation is represented by a Presburger formula $\mathsf{XPres}_{\mathcal{V} \cup \mathcal{V}'}$ where $\mathcal{V}$ and $\mathcal{V}'$ are the current and next variables, respectively.

Model checking techniques require the computation of existential or universal predecessors or successors of a given set of states. The definition of existential and universal preimages respectively images is given below:

Definition 22 (Predecessors and Successors)**.** *Let $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S}$ be a transition relation and Q be a subset of $\mathcal{S}$, $Q \subseteq \mathcal{S}$. The existential and universal preimage of sets of states are defined as follows:*

$$\mathsf{pre}_{\exists}^{\mathcal{R}}(Q) := \{s_1 \in \mathcal{S} \mid \exists s_2.(s_1, s_2) \in \mathcal{R} \wedge s_2 \in Q\}$$
$$\mathsf{pre}_{\forall}^{\mathcal{R}}(Q) := \{s_1 \in \mathcal{S} \mid \forall s_2.(s_1, s_2) \in \mathcal{R} \rightarrow s_2 \in Q\}$$

The existential and universal image are defined similarly by

$$\mathsf{suc}_{\exists}^{\mathcal{R}}(Q) := \{s_2 \in \mathcal{S} \mid \exists s_1.(s_1, s_2) \in \mathcal{R} \wedge s_1 \in Q\}$$
$$\mathsf{suc}_{\forall}^{\mathcal{R}}(Q) := \{s_2 \in \mathcal{S} \mid \forall s_1.(s_1, s_2) \in \mathcal{R} \rightarrow s_1 \in Q\}$$

The result of an existential preimage is the set of states that has at least one predecessor in Q whereas the universal preimage is the set of states that have all predecessors in Q. For the existential respectively the universal image we obtain the set of states that has at least one predecessor respectively all predecessor states in Q.

For existential and universal images resp. preimages hold the following duality laws:

Theorem 8 (Duality Laws)**.** *Let $\mathcal{R}$ be the transition relation and $Q \subseteq \mathcal{S}$ a subset of the set of states as defined before. Then the following equations hold:*

$$\mathsf{pre}_{\exists}^{\mathcal{R}}(Q) = \mathcal{S} \backslash \mathsf{pre}_{\forall}^{\mathcal{R}}(\mathcal{S} \backslash Q)$$
$$\mathsf{pre}_{\forall}^{\mathcal{R}}(Q) = \mathcal{S} \backslash \mathsf{pre}_{\exists}^{\mathcal{R}}(\mathcal{S} \backslash Q)$$
$$\mathsf{suc}_{\exists}^{\mathcal{R}}(Q) = \mathcal{S} \backslash \mathsf{suc}_{\forall}^{\mathcal{R}}(\mathcal{S} \backslash Q)$$
$$\mathsf{suc}_{\forall}^{\mathcal{R}}(Q) = \mathcal{S} \backslash \mathsf{suc}_{\exists}^{\mathcal{R}}(\mathcal{S} \backslash Q)$$

Another important property of images and preimages is monotony.

Theorem 9 (Monotonicity Laws)**.** *Given are two sets of states Q_1 and Q_2 with $Q_1 \subseteq Q_2$. Then, for the existential and universal image resp. preimage holds:*

$$\mathsf{pre}_{\exists}^{\mathcal{R}}(Q_1) \subseteq \mathsf{pre}_{\exists}^{\mathcal{R}}(Q_2)$$
$$\mathsf{pre}_{\forall}^{\mathcal{R}}(Q_1) \subseteq \mathsf{pre}_{\forall}^{\mathcal{R}}(Q_2)$$
$$\mathsf{suc}_{\exists}^{\mathcal{R}}(Q_1) \subseteq \mathsf{suc}_{\exists}^{\mathcal{R}}(Q_2)$$
$$\mathsf{suc}_{\forall}^{\mathcal{R}}(Q_1) \subseteq \mathsf{suc}_{\forall}^{\mathcal{R}}(Q_2)$$

A very powerful description language and base logic for many verification methods is the μ-calculus. A formula φ describes the system behavior that has to be proved within a given Kripke structure $\mathcal{K}$. Formulas of extended Presburger arithmetic are able to express particular stages of the computation of a system but can not be used to reason about its temporal behavior. The μ-calculus extends any given base logic by introducing modal and fixpoint operators such that the extended Presburger arithmetic XPres has the corresponding temporal extension XPres^{μ}, [19].

Definition 23 (Syntax of XPres^{μ})**.** *For the set of extended Presburger formulas* XPres_{ν}*, the set* XPres^{μ} *of Presburger* μ*-calculus formulas is the smallest set satisfying the following rules:*

- $\mathsf{XPres}_{\nu} \subseteq \mathsf{XPres}^{\mu}$
- $\neg\varphi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi \in \mathsf{XPres}^{\mu}$
- $\varphi \wedge \psi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi, \psi \in \mathsf{XPres}^{\mu}$
- $\varphi \vee \psi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi, \psi \in \mathsf{XPres}^{\mu}$
- $\Diamond\varphi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi \in \mathsf{XPres}^{\mu}$
- $\Box\varphi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi \in \mathsf{XPres}^{\mu}$
- $\mu x.\varphi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi \in \mathsf{XPres}^{\mu}$ *and that all occurrences of* x *in* φ *are positive*
- $\nu x.\varphi \in \mathsf{XPres}^{\mu}$ *provided that* $\varphi \in \mathsf{XPres}^{\mu}$ *and that all occurrences of* x *in* φ *are positive*

The semantics of the Presburger μ-calculus is defined recursively as the set of states that satisfy a formula φ which is denoted by $[\![\varphi]\!]_{\mathcal{K}}$.

Definition 24 (Semantics of XPres^{μ})**.** *Given an IKS* $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$*, the semantic of a formula* $\Phi \in \mathsf{XPres}^{\mu}$ *as a subset of* $\mathcal{S}$ *is recursively defined as:*

$[\![\varphi]\!]_{\mathcal{K}} :\equiv [\![\varphi]\!] \cup \mathcal{S}$ *for any* $\varphi \in \mathsf{XPres}_{\nu}$
$[\![\neg\varphi]\!]_{\mathcal{K}} :\equiv \mathcal{S} \setminus [\![\varphi]\!]_{\mathcal{K}}$
$[\![\varphi \wedge \psi]\!]_{\mathcal{K}} :\equiv [\![\varphi]\!]_{\mathcal{K}} \cap [\![\psi]\!]_{\mathcal{K}}$ $\quad [\![\varphi \vee \psi]\!]_{\mathcal{K}} :\equiv [\![\varphi]\!]_{\mathcal{K}} \cup [\![\psi]\!]_{\mathcal{K}}$
$[\![\Diamond\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{pre}_{\exists}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$ $\quad [\![\Box\varphi]\!]_{\mathcal{K}} :\equiv \mathsf{pre}_{\forall}^{\mathcal{R}}([\![\varphi]\!]_{\mathcal{K}})$
$[\![\mu x.\varphi]\!]_{\mathcal{K}}$ *is the least set* Q *where* $Q = [\![\varphi]\!]_{\mathcal{K}_x^Q}$ *holds*
$[\![\nu x.\varphi]\!]_{\mathcal{K}}$ *is the greatest set* Q *where* $Q = [\![\varphi]\!]_{\mathcal{K}_x^Q}$ *holds*

$\mathcal{K}_x^Q$ *is the IKS that is obtained from* $\mathcal{K}$ *by changing the states* s *of* $\mathcal{K}$ *such that* $s \in Q$ *satisfy* x*. An IKS* $\mathcal{K} = (\mathcal{S}, \mathcal{I}, \mathcal{R})$ *satisfies a formula* Φ *iff* $\mathcal{I} \subseteq [\![\Phi]\!]_{\mathcal{K}}$ *holds.*

Let the formula $\Phi_Q(x)$ represent the set of states Q. Then the existential image can be computed symbolically with the following formula:

$$[\exists x.\mathcal{R}(x,y) \wedge \Phi_Q(x)]^x_y \tag{1}$$

$\mathcal{R}(x,y) \wedge \Phi_Q(x)$ represents the set of transitions leaving Q. The quantifier removes the source states of the transition relation such that only the successors remain in the set.
The existential preimage can be computed by $\exists y.\mathcal{R}(x,y) \wedge [\Phi_Q(x)]^y_x$. All transitions that lead to Q are represented by $\mathcal{R}(x,y) \wedge [\Phi_Q(x)]^y_x$. The quantifier eliminates the target states and the remaining states are the predecessors of Q.
The universal preimage respectively image can be obtained by using the duality laws given in theorem 8.

For BDDs Somenzi proves in [22] that the heuristic functions constrain and restrict can be used efficiently for image computation. These functions simplify the transition relation in order to reduce the size of the corresponding BDD by using don't care information. Within the transition system this means that transitions from or to unreachable states can be used to reduce the BDD that represents the resulting formula. The results of fixpoint computation restricted to reachable states do not change.

Since the representation of infinite data structures requires automata it would be interesting to know if the restrict operator can be used for image computation as well in order to minimize the automaton.

Lemma 2. *For two given automata $\mathcal{A}$ and $\mathcal{B}$ holds the following equation:*
$\mathcal{L}((\mathcal{A} \downarrow \mathcal{B}) \wedge \mathcal{B}) = \mathcal{L}(\mathcal{A} \wedge \mathcal{B})$

Proof. Using the definition the following equation has to be valid:

$$
\begin{array}{lr}
(\mathcal{B} \rightarrow (\mathcal{A} \leftrightarrow \mathcal{A}')) \rightarrow ((\mathcal{B} \wedge \mathcal{A}') \leftrightarrow (\mathcal{A} \wedge \mathcal{B})) & \Leftrightarrow \\
\neg(\neg\mathcal{B} \vee (\mathcal{A} \leftrightarrow \mathcal{A}')) \vee (\mathcal{A} \wedge \mathcal{A}' \wedge \mathcal{B} \vee (\neg(\mathcal{A}' \wedge \mathcal{B}) \wedge \neg(\mathcal{A} \wedge \mathcal{B}))) & \Leftrightarrow \\
\mathcal{B} \wedge (\mathcal{A} \oplus \mathcal{A}') \vee (\mathcal{A} \wedge \mathcal{A}' \wedge \mathcal{B} \vee ((\neg\mathcal{A}' \vee \neg\mathcal{B}) \wedge (\neg\mathcal{A} \vee \neg\mathcal{B})) & \Leftrightarrow \\
(\mathcal{A} \wedge \neg\mathcal{A}' \wedge \mathcal{B}) \vee (\neg\mathcal{A} \wedge \mathcal{A}' \wedge \mathcal{B}) \vee & \\
(\mathcal{A} \wedge \mathcal{A}' \wedge \mathcal{B} \vee (\neg\mathcal{A} \wedge \neg\mathcal{A}') \vee (\neg\mathcal{A}' \wedge \mathcal{B}) \vee (\neg\mathcal{A} \wedge \neg\mathcal{B}) \vee \neg\mathcal{B}) & \Leftrightarrow \\
\mathcal{B} \wedge ((\mathcal{A} \wedge \neg\mathcal{A}') \vee (\neg\mathcal{A} \wedge \mathcal{A}') \wedge (\mathcal{A} \wedge \mathcal{A}') \vee \neg\mathcal{A}' \vee \neg\mathcal{A}) \vee \neg\mathcal{B} & \Leftrightarrow \\
\mathcal{B} \vee \neg\mathcal{B} & \Leftrightarrow \\
\mathsf{true} &
\end{array}
$$

□

The property that $\mathcal{L}((\mathcal{A} \downarrow \mathcal{B}) \wedge \mathcal{B}) = \mathcal{L}(\mathcal{A} \wedge \mathcal{B})$ holds can now be used for image computation.

Definition 25. *The existential preimage and the existential image that are computed with the restrict operator are defined as:*

$$\mathsf{pre}_{\downarrow\exists}^{\mathcal{R}}(Q) = \exists y.\mathcal{R}(x,y) \wedge [\Phi_Q(x)]_x^y$$
$$\mathsf{suc}_{\downarrow\exists}^{\mathcal{R}}(Q) := [\exists x.\mathcal{R}(x,y) \downarrow \Phi_Q(x)]_y^x$$

The universal preimage and the universal image can be computed by using the duality laws (Theorem 8). A fixpoint formula ψ that is evaluated with the restrict operator is denoted as $[\![\psi]\!]_{\downarrow\mathcal{K}}$.

The result of an intersection of two languages is always smaller or equal to both languages such that $\mathcal{L}(\mathcal{A} \downarrow \mathcal{B}) \cap \mathcal{L}(\mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \downarrow \mathcal{B})$. Therefore, we get that $\mathcal{L}(\mathcal{A} \wedge \mathcal{B}) \subseteq \mathcal{L}(\mathcal{A} \downarrow \mathcal{B})$. Since quantifiers are monotonic and the automata $\mathcal{A}$ and $\mathcal{B}$ are Presburger formulas that can represent the transition relation and a set of states, we obtain the following inequation:

$$\mathsf{suc}_{\exists}^{\mathcal{R}}(Q) \subseteq \mathsf{suc}_{\downarrow\exists}^{\mathcal{R}}(Q). \tag{2}$$

The converse does not necessarily hold. We have at least an overapproximation of the exact image. Nevertheless, the problem whether a state belongs to the set of the successor states is semi-decidable such that $s \notin \mathsf{suc}_{\downarrow\exists}^{\mathcal{R}}(Q) \Rightarrow s \notin \mathsf{suc}_{\exists}^{\mathcal{R}}(Q)$.

Theorem 10 (Subsetrelations of Preimages and Images)**.** *Let* $\mathsf{pre}_{\exists}^{\mathcal{R}}$, $\mathsf{pre}_{\forall}^{\mathcal{R}}$, $\mathsf{suc}_{\exists}^{\mathcal{R}}$ *and* $\mathsf{suc}_{\forall}^{\mathcal{R}}$ *be the preimages and images that are computed with the conjunction and* $\mathsf{pre}_{\downarrow\exists}^{\mathcal{R}}$, $\mathsf{pre}_{\downarrow\forall}^{\mathcal{R}}$, $\mathsf{suc}_{\downarrow\exists}^{\mathcal{R}}$ *and* $\mathsf{suc}_{\downarrow\forall}^{\mathcal{R}}$ *the preimages and images that use the restrict operator, respectively. Then, the following subsetrelations hold:*

$$\mathsf{pre}_{\exists}^{\mathcal{R}}(Q) \subseteq \mathsf{pre}_{\downarrow\exists}^{\mathcal{R}}(Q).$$
$$\mathsf{pre}_{\forall}^{\mathcal{R}}(Q) \subseteq \mathsf{pre}_{\downarrow\forall}^{\mathcal{R}}(Q).$$
$$\mathsf{suc}_{\exists}^{\mathcal{R}}(Q) \subseteq \mathsf{suc}_{\downarrow\exists}^{\mathcal{R}}(Q).$$
$$\mathsf{suc}_{\forall}^{\mathcal{R}}(Q) \subseteq \mathsf{suc}_{\downarrow\forall}^{\mathcal{R}}(Q).$$

Since we know the relation between images computed with conjunction and images computed with the restriction it is important how the restriction influences the results of fixpoint computations.

Theorem 11. *Given are two fixpoint iterations $[\![\varsigma x.\varphi]\!]_{\mathcal{K}}$ with the common image or preimage computation and $[\![\varsigma x.\varphi]\!]_{\downarrow\mathcal{K}}$ that uses the restrict operator to obtain predecessor or successor states and $\varsigma \in \{\mu, \nu\}$. Then the following subsetrelations hold:*

$$1)[\![\mu x.\varphi]\!]_{\mathcal{K}} \subseteq [\![\mu x.\varphi]\!]_{\downarrow\mathcal{K}}$$
$$2)[\![\nu x.\varphi]\!]_{\downarrow\mathcal{K}} \subseteq [\![\nu x.\varphi]\!]_{\mathcal{K}}$$

Proof.

1. $\underline{[\![\mu x.\varphi]\!]_{\mathcal{K}} \subseteq [\![\mu x.\varphi]\!]_{\downarrow\mathcal{K}}}$

   A least fixpoint is computed by the following iteration that is given by the Tarski-Knaster theorem for the μ-calculus:

   $$Q_0 := \emptyset \qquad Q_{i+1} := [\![\varphi]\!]_{\mathcal{K}_x^{Q_i}}.$$

   There is a number $n \in \mathbb{N}$ such that $Q_n = Q_{n+1}$ where we have $Q_n = [\![\mu x.\varphi]\!]_{\mathcal{K}}$. For the computation of a least fixpoint the sequence of sets Q_i is monotonic and increasing $Q_0 \subset Q_1 \subset ... \subset Q_n = Q_{n+1}$.

   Let Q_i be the computed sets of the fixpoint computation with the common preimage computation and R_j the sets of the fixpoint computation with the preimage computed with the restrict operator. Assume, that the fixpoint computation is final, that means, there exist natural numbers $n, m \in \mathbb{N}$ such that $Q_n = Q_{n+1}$ and $R_m = R_{m+1}$. Due to the fact that both functions are continuous and monotonic it holds that:

   $$Q_0 \subset Q_1 \subset ... \subset Q_n = Q_{n+1}$$
   $$R_0 \subset R_1 \subset ... \subset R_m = R_{m+1}$$

   The initial sets of states are equal since both computations start with the empty set. Therefore, it holds $Q_0 \subseteq R_0$.

   For each iteration step k it holds due to the monotonicity of state transformers [19], the monotonicity laws of temporal operators (Theorem 9) and the subsetrelations of preimages and images with common computation and the ones computed with the restrict operator (Theorem 10) $Q_k \subseteq R_k$.

   For least fixpoints we obtain $[\![\mu x.\varphi]\!]_{\mathcal{K}} \subseteq [\![\mu x.\varphi]\!]_{\downarrow\mathcal{K}}$.

2. $\llbracket \nu x.\varphi \rrbracket_{\downarrow\mathcal{K}} \subseteq \llbracket \nu x.\varphi \rrbracket_{\mathcal{K}}$

The computation of greatest fixpoints is given by the Tarski-Knaster theorem that defines the iteration as given below:

$$Q_0 := \mathcal{S} \qquad Q_{i+1} := \llbracket \varphi \rrbracket_{\mathcal{K}_x^{Q_i}}.$$

There exists a number $n \in \mathbb{N}$ such that $Q_n = Q_{n+1}$ where we have $Q_n = \llbracket \nu x.\varphi \rrbracket_{\mathcal{K}}$. For the computation of a greatest fixpoint the sequence of sets Q_i is monotonic and decreasing $Q_0 \supset Q_1 \supset ... \supset Q_n = Q_{n+1}$.

Let again Q_i be the computed sets of the fixpoint computation with the common preimage computation and R_j the sets of the fixpoint computation with the preimage computed with the restrict operator. Suppose, that the fixpoint computation is final, that means there exist natural numbers $n, m \in \mathbb{N}$ such that $Q_n = Q_{n+1}$ and $R_m = R_{m+1}$. Due to the fact that both functions are continuous and monotone it holds that:
$Q_0 \supset Q_1 \supset ... \supset Q_n = Q_{n+1}$
$R_0 \supset R_1 \supset ... \supset R_m = R_{m+1}$

The initial sets of states start with the set of all states $\mathcal{S}$ and therefore, we obtain $Q_0 \supseteq R_0$.

For each iteration step k it holds due to the monotonicity of state transformers [19], the monotonicity laws of temporal operators (Theorem 9) and the subsetrelations of preimages and images with common computation and the ones computed with the restrict operator (Theorem 10) $Q_k \supseteq R_k$.

Then, for greatest fixpoints it holds that $\llbracket \nu x.\varphi \rrbracket_{\downarrow\mathcal{K}} \subseteq \llbracket \nu x.\varphi \rrbracket_{\mathcal{K}}$.

□

The result for fixpoint computation with the restrict operator is an overapproximation of least fixpoints and an underapproximation of greatest ones. The properties can be used as semi-decisions whether the set of initial states lies within the computed sets: If the set of initial states is not in the overapproximation, it is neither in the exact computed set of states: $\mathcal{I} \not\subseteq \llbracket \mu x.\varphi \rrbracket_{\downarrow\mathcal{K}} \Rightarrow \mathcal{I} \not\subseteq \llbracket \mu x.\varphi \rrbracket_{\mathcal{K}}$.
If the set of initial states lies in the underapproximation, it lies in the exact solution as well: $\mathcal{I} \subseteq \llbracket \nu x.\varphi \rrbracket_{\downarrow\mathcal{K}} \Rightarrow \mathcal{I} \subseteq \llbracket \nu x.\varphi \rrbracket_{\mathcal{K}}$.

4 The Restrict Algorithm

The restrict algorithm minimizes a given automaton $\mathcal{A}_1$ w.r.t. to a restricting automaton $\mathcal{A}_2$ by computing the minimal number of equivalence classes of their product. States of the product automaton are constructed of state pairs of $\mathcal{A}_1$ and $\mathcal{A}_2$ that are reachable from the initial states with the same input. The product automaton contains three possible kinds of states. Either the states belong to the set of final or non-final states or they have an unspecified acceptance behavior. The goal of the algorithm is to assign the don't care states such that the number of equivalence classes is minimal. The size of an automaton is directly dependent on the number of equivalence classes since all states that belong to the same class can be merged into a single state.

The algorithm consists of the following steps:

- Construction of the product automaton
- Computing the equivalence relation
- Counting the number of equivalence classes
- Minimization with binary search
- Formula evaluation and state assignment
- Creating the minimized automaton

4.1 Construction of the Product Automaton

The function requires two automata as parameters that are used to construct the product automaton. Additionally it gives back a list of states that belong to the set of don't care states. At the beginning the transition relation δ, the labeling function λ, the set of final states $\mathcal{F}$ and the set of don't care states DC of the product automaton are empty. The initial state of the product is the state pair of the initial states of both automata. A list (ToDo) which contains all statepairs that have to be computed and the set of states of the product automaton $\mathcal{Q}$ are initialized with this initial state pair.

Starting with the initial state, the successors of every state pair in the list are computed. For every transition of both states the conjunction of their input variables is evaluated. If it is empty the new state pair is not reachable and nothing happens. If not, the labeling function and the transitions are added to the product automaton. Pairs of reached successors that already have been traversed can be found in the state set of the product automaton. All state pairs that are reached by the transition for the first time create a new state of the product automaton and have to be inserted into the product state set, the ToDo list and probably into the set of final or don't care states.

States of the product automaton belong to the set of final states $\mathcal{F}$ if for the statepairs holds $(q_1, q_2) \in \mathcal{F}_1 \times \mathcal{F}_2$. The don't care set consists of statepairs that are composed as follows $(q_1, q_2) \in \mathcal{Q}_1 \times (\mathcal{Q}_2 \backslash \mathcal{F}_2)$.

After checking all successor states, the first state of the list is deleted and the method proceeds with the other state pairs in the list until it is empty. The result is the product automaton with states corresponding to one of three state sets.

```
function Product(A1, A2) : (A, DC)

  q0 := (q0^1, q0^2);
  λ := ∅;
  δ := ∅;
  Q := (q0^1, q0^2);
  F := ∅;
  DC := ∅;
  ToDo := (q0^1, q0^2);

  repeat
    (q1, q2) := head(ToDo);
    forall (q1' ∈ suc_A1(q1))
      forall (q2' ∈ suc_A2(q2))
        l := λ1(q1, q1') ∧ λ2(q2, q2');
        if (l ≢ 0)then
          λ := λ ∪ {((q1, q2), (q1', q2')), l};
          δ := δ ∪ {((q1, q2), (q1', q2'))};
          if ((q1', q2') ∉ Q) then
            ToDo := ToDo ∪ {(q1', q2')};
            Q := Q ∪ {(q1', q2')};
            if (q2' ∉ F2) then DC = DC ∪ (q1', q2');
            elseif (q1' ∈ F1) then F = F ∪ (q1', q2');
          endif;
        endif;
      endfor;
    endfor;
    ToDo = ToDo\(q1, q2);
  until (ToDo = ∅);

  return (A, DC);
end.
```

Figure 6: Computing the product automaton

4.2 Computing the Equivalence Relation

In order to minimize the product automaton the greatest fixpoint of the equivalence relation E has to be computed. Based on the equivalence relation the assignment of different states to equivalence classes is ascertainable. The result of the method is a temporary equivalence relation since the assigment of don't care states to the set of final or non-final states can not be decided yet. The method uses a layerwise computation of the equivalence relation based on the algorithm presented in [25].

For the initialization of E the set of equivalent states the acceptance behavior of each possible state pair is evaluated. States are equal if they both belong to the set of final states $\mathcal{F}$ or non-final states $\mathcal{Q}\backslash\mathcal{F}$. For each state pair we obtain a Boolean function that tells us whether the pair is equivalent or not. A final state has the value true, a non-final state the value false. For don't care states the accepting behavior is unknown yet. Therefore we introduce one variable for each don't care state. The variables are evaluated later in order to reduce the number of equivalence classes.

$$\mathsf{Init}(q) := \begin{cases} v_q & \textbf{if } (q \in DC) \\ 1 & \textbf{if } (q \in \mathcal{F}) \\ 0 & \textbf{if } (q \in \mathcal{Q}\backslash(\mathcal{F} \cup DC)) \end{cases}$$

The algorithm EquivStates starts with all possible state pairs of the product automaton. The initial set of equivalent states is the set of the reflexive states and the states which have the same acceptance behavior. State pairs without don't care states can be evaluated immediately to a constant formula true or false. The remaining state pairs return a Boolean formula.

Now the greatest fixpoint can be computed with successive approximations. For each state pair and all its successor states it has to be checked if they belong to the equivalence relation. There are three possibilities:

- $\mathsf{E_k}(q_i, q_j) = 0$:
  state pairs are not equivalent if they don't have the same acceptance behavior or there exists a path to a state pair which is not equivalent

- $\mathsf{E_k}(q_i, q_j) = 1$:
  state pairs are equivalent if they have the same acceptance behavior and all paths lead to equivalent states.

- $\mathsf{E_k}(q_i, q_j)$ is a formula:
  state pairs return a formula if they consist either of at least one don't care state or if they are equivalent and lead to such a pair (or both).

This layerwise computation terminates if there is a number f such that $\mathsf{E_f} = \mathsf{E_{f+1}}$ holds. The maximal number of iteration steps is bounded by the diameter d of the automaton which is given by the longest distance from a state to the initial state.

In the next section, the number of equivalence classes is counted. Since every state that is distinguishable from all the other states belongs to a different equivalence class the number of distinguishable states equals the number of required classes.

Theorem 12 (Equivalence of States)**.** *Let n be the number of the states in the product automaton. A state q_i is distinguishable to another state q_j if the following formula is not satisfiable:*

$$\mathsf{equiv}(q_i, q_j) = (\bigwedge_{k=0}^{n-1} (\mathsf{E}(q_i, q_k) \leftrightarrow \mathsf{E}(q_j, q_k))) \tag{3}$$

Proof. If q_i and q_j are equivalent the corresponding state pairs $\mathsf{E}(q_i, q_j)$ and $\mathsf{E}(q_j, q_i)$ return the constant formula true. Since reflexive pairs are tautologies it holds that $\mathsf{E}(q_i, q_j) = \mathsf{E}(q_j, q_j)$ and $\mathsf{E}(q_j, q_j) = \mathsf{E}(q_j, q_i)$.
For the other state pairs holds due to transitivity of an equivalence relation:

- If q_i and q_k are equal it follows that q_j and q_k are equal as well:
  $\mathsf{E}(q_i, q_k) = \mathsf{true} = \mathsf{E}(q_j, q_k)$

- If q_i and q_k are not equal q_j and q_k are not equal neither:
  $\mathsf{E}(q_i, q_k) = \mathsf{false} = \mathsf{E}(q_j, q_k)$

The result of the conjunction with only true values is true. □

function $\mathsf{EquivStates}(\mathcal{A}, DC) : (\mathcal{Q} \times \mathcal{Q}) \to (\mathbb{B}^n \to \mathbb{B})$

$\mathsf{E} := \emptyset;$

forall $(q_1, q_2) \in \mathcal{Q} \times \mathcal{Q}$
$\mathsf{E} := \mathsf{E} \cup \{((q_1, q_2), \mathsf{Init}(q_1) \leftrightarrow \mathsf{Init}(q_2))\};$
endfor;

repeat
$\mathsf{E_{old}} := \mathsf{E};$
$\mathsf{E} := \emptyset;$

forall $(q_1, q_2) \in \mathcal{Q} \times \mathcal{Q}$
$\mathsf{t} := \mathsf{E_{old}}(q_1, q_2);$
forall $(q_1' \in \mathsf{suc}_{\mathcal{A}}(q_1))$
forall $(q_2' \in \mathsf{suc}_{\mathcal{A}}(q_2))$
if $(\lambda(q_1, q_1') \wedge \lambda(q_2, q_2') \not\equiv 0)$ **then** $\mathsf{t} := \mathsf{t} \wedge \mathsf{E_{old}}(q_1', q_2');$
endfor;
endfor;
$\mathsf{E} := \mathsf{E} \cup \{(q_1, q_2), \mathsf{t}\};$
endfor;

until $\mathsf{E_{old}} = \mathsf{E};$

return $\mathsf{E};$
end.

Figure 7: Algorithm to compute equivalent states

4.3 Counting the Number of Equivalence Classes

The temporary equivalence relation is given to a method that uses the Boolean formulas of the state pairs to count the number of the resulting equivalence classes. For each state the number of equivalence classes increases when the state is distinguishable to all the previous states. Since there is at least one equivalence class the first state is set to true. Then iteratively the following $n-1$ states are compared to be equal to the previous ones. Note that adding the i-th formula to the previous ones means that the result contains $(i+1)$ input values.

A state is equivalent to one of the former states if the following formula is satisfied:

$$\mathsf{Equiv}(q_i) = \bigvee_{j=0}^{i-1} \mathsf{equiv}(q_i, q_j) = \bigvee_{j=0}^{i-1} \Big(\bigwedge_{k=0}^{n-1} (\mathsf{E}(q_i, q_k) \leftrightarrow \mathsf{E}(q_j, q_k)) \Big) \quad (4)$$

Only distinguishable states create a new equivalence class that is why the input of the counter are the inversion of formula (9). The function that computes the input formulas is shown in figure 8. For the i-th state that is compared to the previous states the input formula is given by $x_i = \neg\mathsf{Equiv}(q_i)$. The acceptance behavior of don't care states is still unknown so the result of the counter consists of Boolean formulas. The output formulas represent the number of equivalent classes given as a binary number $\sum_{i=0}^{m} f_i * 2^i$. These functions have to be evaluated afterwards in order to minimize the number of equivalent classes.

---

function $\mathsf{Input}(\mathsf{E}, i) : (\mathbb{B}^n \to \mathbb{B})$

$$\mathsf{I}(q_i) = \bigvee_{l=0}^{i-1} \bigwedge_{k=0}^{n-1} (\mathsf{E}(q_l, q_k) \leftrightarrow \mathsf{E}(q_i, q_k));$$

return $\neg\mathsf{I}(q_i)$;
end.

Figure 8: Counter Input

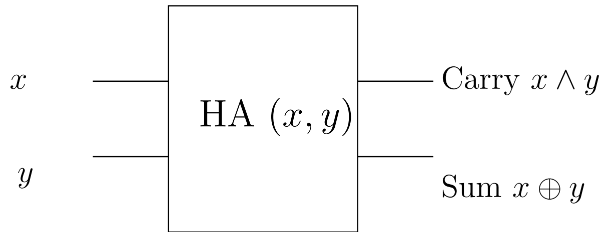

Figure 9: Half-Adder

A counter is a cascade of half adders that adds iteratively a new input value to the previous result (see figure 11). For constant formulas there are two possibilities. If the new input value is false the result remains the same, if it is true the number increases. Every iteration step can be seen as a layerwise computation due to the structure of the counter.

The smallest unit of the counter are the half adders (see figure 9). It has two inputs x and y for the values that have to be summed and two outputs, one for the sum bit s and the other for the carry bit c. The addition of binary numbers can be realized by two Boolean functions. The sum is computed by $s = x \oplus y$, the carry value by $c = x \wedge y$.
As an example, the first state formula is given by true and the second input is set to false since we have only one equivalence class that has to be counted yet. Then the sum bit equals $s = 1 \oplus 0 = 1$ and the carry bit has to be $c = 1 \wedge 0 = 0$.

---

function HA$(s_1, s_2) : ((\mathbb{B}^n \rightarrow \mathbb{B}), (\mathbb{B}^n \rightarrow \mathbb{B}))$

$S := s_1 \oplus s_2;$
$C := s_1 \wedge s_2;$

return $(S, C);$
end.

Figure 10: Function of the half-adder

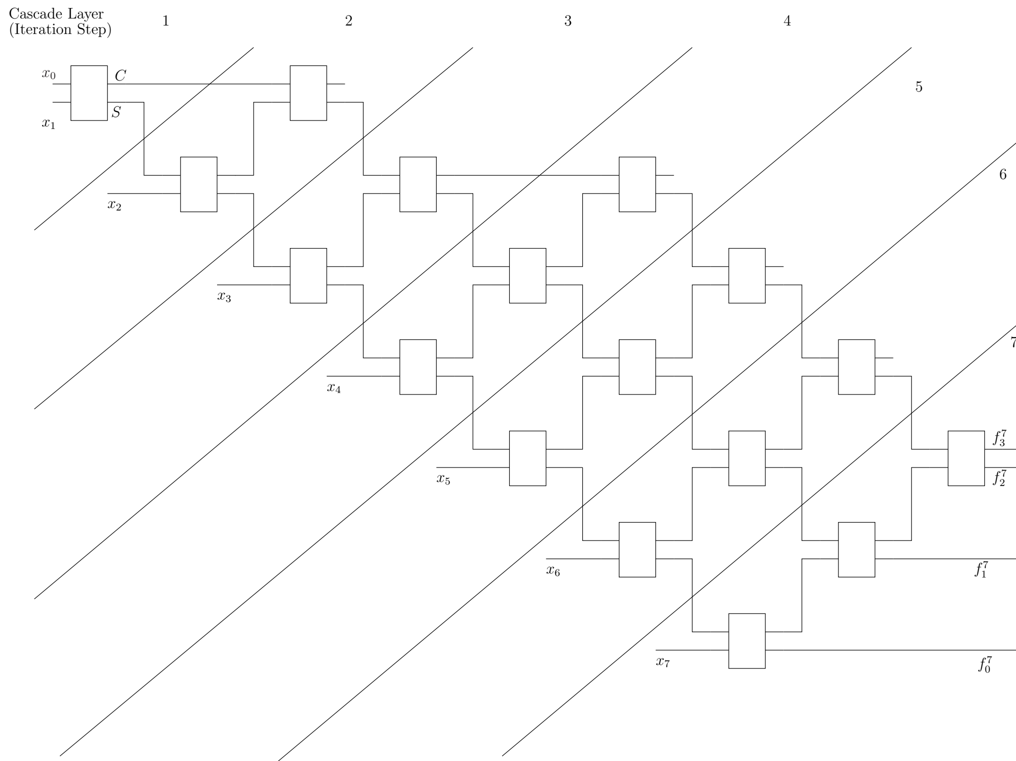

Figure 11: Counter for eight variables

In contrast to the method, the half adder of the first layer shown in figure 11 gets input values of two states. In the algorithm in every layer only one state formula is added such that x_0 normally would be the sum bit of the half adder that creates the first equivalence class (the first state value is set to true and the second input is constant false).

Each layer of the cascade requires a certain number of half adders since the number of output functions depends on the number of input values. To represent a the sum of $(i+1)$ values the most significant bit has the index $m = \lfloor \log_2(i+1) \rfloor$. The number of the required half adders in the i-th layer that are necessary to compute the $m + 1$ output functions is given by $\lceil \log_2(i + 1) \rceil$.

The dependancy of the number of input values, half adders, the MSB-index and the number of output functions can be seen in the next table:

Input values	$(i+1)$	1	2	3	4	5	6	7	8	...
Half-Adders	$\lvert\mathsf{HA}\rvert = \lceil \log_2 (i+1) \rceil$	1	1	2	2	3	3	3	3	...
Index of MSB	$m = \lfloor \log_2 (i+1) \rfloor$	0	1	1	2	2	2	2	3	...
Output functions	$\lvert f\rvert = m+1$	1	2	2	3	3	3	3	4	...

Now we can distinguish between two cases:

- $|(i+1)| \neq 2^r, r \in \mathbb{N}$
  In most cases the number of the input values does not equal 2^r. Then it is $\lceil \log_2(i+1) \rceil = \lfloor \log_2(i+1) \rfloor + 1$, which means that the number of the required half adders in the i-th layer equals the number of output functions. The sum bit of each half adder returns a value of the output function.

- $|(i+1)| = 2^r$
  If the number of the input values equals 2^r the number of the resulting functions is bigger than the number of the half adders used in the layer because $\lceil \log_2(i+1) \rceil = \lfloor \log_2(i+1) \rfloor$ holds. In this special case the function with the most significant bit is available at the carry output of the last half adder (compare figure 11,layer 1 and 3)

```
function Counter(E) : (𝔹^n → 𝔹))

   COut := ∅;
   (f_0, C) := HA(true, false);
   for (i := 1 to n − 1) do
      m := ⌊log_2(i + 1)⌋;
      for (j := 0 to m) do
         if (j = m) ∧ (m = ⌈log_2(i + 1)⌉)then f_m := C;
         else
            if (j = 0) then C := Input(E, i);
            (f_j, C) := HA(C, f_j);
         endif;
      endfor;
   endfor;
   COut := ⋃_{j=0}^{m} f_j;
   return(COut);
end.
```

Figure 12: Counter

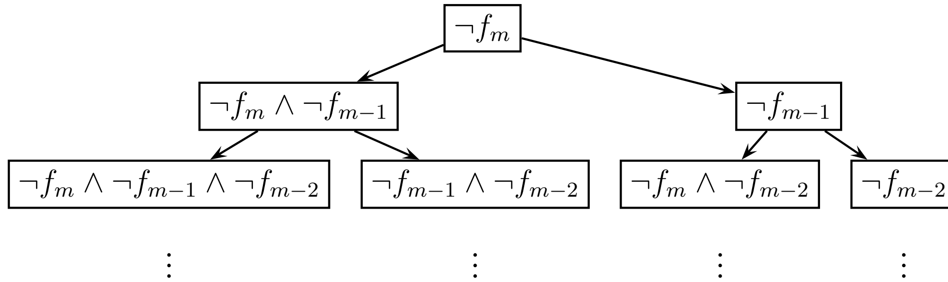

Figure 13: Binary search

4.4 Evaluating the Counter Result

After the counter computed the resulting formulas binary search is used to optimize the number of equivalent classes. Since we are interested in the smallest automaton the minimal number of distinguishable states has to be found. Starting with the most significant bit the method checks whether the negated formula (which represents a false value in order to minimize the number of equivalence classes) has a model. If this is the case, the index of the formula is decreased and the satisfiability of the conjunction of both negated formulas is checked. If the first formula f_m is unsatisfiable, the negated formula f_{m-1} has to be evaluated. In figure 13 the composition of the minimizing formula is shown in a binary tree. The nodes represent the satisfiability check of the formula. The root (layer 0) contains only the negated function with the most significant bit. In each layer k a model of the conjunction of the previous satisfiable formula and the negation of the current formula f_{m-k} is searched. Child nodes on the left are the composed formulas when the formula of the parent node is satisfiable, the right node is the resulting formula when the parent node is contradictional.

```
function BinSearch(COut) : (𝔹^n → 𝔹)

   One := 1;

   for (k = m  to  0) do
      if (One ∧ ¬f_k ≢ 0) then One := One ∧ ¬f_k;
   endfor;

   return One;
end.
```

Figure 14: Searching optimal model

4.5 State Assignment

The algorithm of binary search returns a formula that represents the minimal number of equivalence classes. The model that satisfies this formula maps each introduced variable on a Boolean value. Due to the fact that every variable replaces a don't care state the evaluation provides us an unambiguous assigment of states with unspecified acceptance behavior to either the set of final states or the set of non-final states. Variables that are evaluated as true assign the corresponding don't care state into the set of final states whereas false variables represent states belonging to the set of non-final states. At least the non-constant formulas of the equivalence relation can be evaluated with the given model. Each state pair that is evaluated as false is distinguishable and all state pairs that are true are member of the same equivalence class.

---

function Assign(One) : $\mathcal{F}$

  temp := Model(One);
  forall $(q \in DC)$
    if $(\mathsf{temp}(v_q) \equiv 1)$ **then** $\mathcal{F} := \mathcal{F} \cup q$;
  endfor;
  return $\mathcal{F}$;
end.

Figure 15: Assign values to DC states

4.6 Minimization

In the last step the equivalence relation is used to construct the new minimized automaton $\mathcal{A}'$. Beginning with one state it is checked whether it belongs to existant equivalence classes. If not a new state is created, otherwise it is marked to which equivalence class the state belongs to. After that computation the minimized automaton obtains as many states as there are equivalence classes due to the fact that all states of one partition are merged to one state. The new transition relation consists of all transitions in the old relation with the difference that the source and target states are now the states that represent the partitions where the old states are contained.

```
function Minimize(E, 𝒜) : 𝒜'

   𝒬' := ∅;        δ' := ∅;
   ℱ' := ∅;        λ' := ∅;
   forall (q_i ∈ 𝒬) do EC(q_i) := ∅;

   forall ((q_1, q_2) ∈ 𝒬 × 𝒬) do
      if (E(q_1, q_2) = true) then
         if (EC(q_1) = ∅) then
            𝒬' := 𝒬' ∪ {q_2};
            if (q_2 ∈ ℱ) then ℱ' := ℱ' ∪ {q_2};
         endif;
      EC(q_2) := EC(q_1);
      endif;
   endfor;

   forall ((q_1, q_2) ∈ δ) do
      δ' := δ' ∪ (EC(q_1), EC(q_2));
      λ' := λ' ∪ (EC(q_1), EC(q_2), l);
   endfor;

   q'_0 := EC(q_0);

   return 𝒜';
end.
```

Figure 16: Construction of the minimized automaton

5 Example

In this example it is shown how the algorithm works with two automata that already have been translated from Presburger arithmetic. The one that has to be restricted represents $(x > 0)$, the restricting one $(x \neq 0)$.

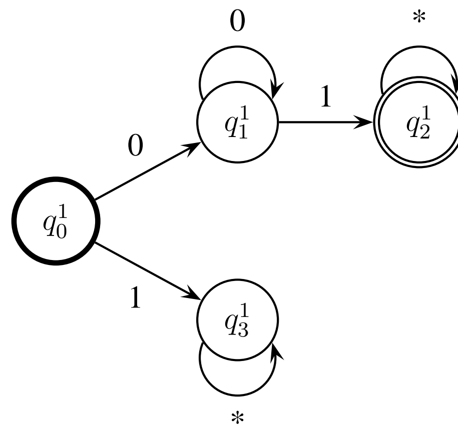

Figure 17: Automaton to be restricted

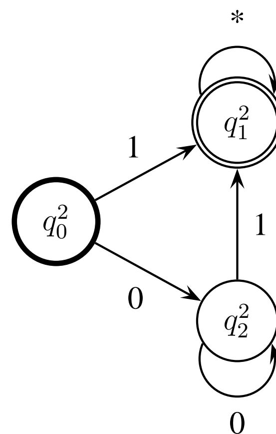

Figure 18: Restricting automaton

After the constrution of their product, the result is an automaton that has one don't care state as it can be seen below. The initial state q_0 is the composition of the initial states q_0^1 and q_0^2. The reachable states of the product automaton are a non-final state $q_1 := (q_3^1, q_1^2)$, a don't care state $q_2 := (q_1^1, q_2^2)$, and a final state $q_3 := (q_2^1, q_1^2)$.

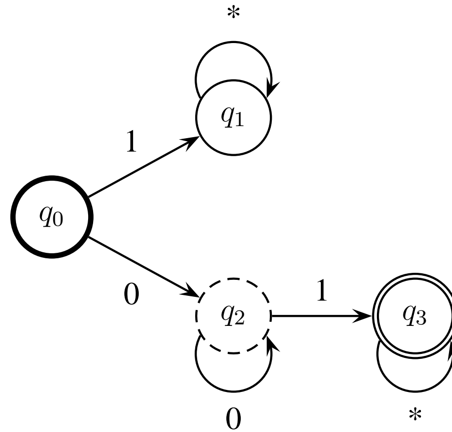

Figure 19: Restricted automaton before selection of final states

Now the equivalence relation is computed. Since the empty word shall not be accepted the initial state is not equal to one of the other states. The initial set E_0 without writing the symmetric state pairs is:

$$\begin{array}{llll} \mathsf{E}_0(q_0,q_0)=1 & & & \\ \mathsf{E}_0(q_0,q_1)=0 & \mathsf{E}_0(q_1,q_1)=1 & & \\ \mathsf{E}_0(q_0,q_2)=0 & \mathsf{E}_0(q_1,q_2)=(0\leftrightarrow v_2) & \mathsf{E}_0(q_2,q_2)=1 & \\ \mathsf{E}_0(q_0,q_3)=0 & \mathsf{E}_0(q_1,q_3)=0 & \mathsf{E}_0(q_2,q_3)=(v_2\leftrightarrow 1) & \mathsf{E}_0(q_3,q_3)=1 \end{array}$$

Since reflexive state pairs are tautologies, they remain in the set until the computations finishes. State pairs that are false do not change their values. For the next iteration step the changed pairs are:

$$\begin{array}{l} \mathsf{E}_1(q_1,q_2)=(0\leftrightarrow v_2)\wedge \mathsf{E}_0(q_1,q_2)\wedge \mathsf{E}_0(q_1,q_3)=0 \\ \mathsf{E}_1(q_2,q_3)=(v_2\leftrightarrow 1)\wedge \mathsf{E}_0(q_2,q_3)\wedge \mathsf{E}_0(q_2,q_2)=(v_2\leftrightarrow 1) \end{array}$$

In E_2 nothing changes any more, it holds $\mathsf{E}_2=\mathsf{E}_1$.

Since there are four states the index of the most significant bit is $\lfloor log_2 4 \rfloor = 2$ and the number of required half-adders is $\lceil log_2 4 \rceil = 2$. The first state input is set to true due to the fact that the initial state is distinguishable to all the other states. The minimized automaton has at least two equivalence classes because the empty word ε shall not be accepted.

The input values of the counter are:

$$x_1 = \neg(\mathsf{Equiv}(q_1)) = \neg(\bigwedge_{k=0}^{3} \mathsf{E}_2(q_0, q_k) \leftrightarrow \mathsf{E}_2(q_1, q_k)) = 1$$

$$x_2 = \neg(\mathsf{Equiv}(q_2)) = \neg(\bigvee_{j=0}^{1} (\bigwedge_{k=0}^{3} \mathsf{E}_2(q_j, q_k) \leftrightarrow \mathsf{E}_2(q_2, q_k))) = 1$$

$$x_3 = \neg\mathsf{Equiv}(q_3) = \neg((\bigvee_{j=0}^{2} (\bigwedge_{k=0}^{3} \mathsf{E}_2(q_j, q_k) \leftrightarrow \mathsf{E}_2(q_3, q_k))) = \neg(1 \leftrightarrow (v_2 \leftrightarrow 1))$$

Computing the second layer, there are two input states that require two half-adders and two output functions:

$$f_0^1 = f_0^0 \oplus x(1) = 1 \oplus 1 = 0$$
$$f_1^1 = f_0^0 \wedge x(1) = 1 \wedge 1 = 1$$

The third layer has three input states, two half-adders and two output functions:

$$f_0^2 = f_0^1 \oplus x(2) = 0 \oplus 1 = 1$$
$$f_1^2 = (f_0^1 \wedge x(2)) \oplus f_1^1 = (0 \wedge 1) \oplus 1 = 1$$

In the last layer the fourth state is added to the former result and the three output functions are the result of two half-adders:

$$f_0^3 = f_0^2 \oplus x(3) = 1 \oplus \neg(1 \equiv (v_2 \equiv 1))$$
$$f_1^3 = (f_0^2 \wedge x(3)) \oplus f_1^2 = (1 \wedge \neg(1 \leftrightarrow (v_2 \leftrightarrow 1))) \oplus 1$$
$$f_2^3 = (f_0^2 \wedge x(3)) \wedge f_1^2 = (1 \wedge \neg(1 \leftrightarrow (v_2 \leftrightarrow 1))) \wedge 1$$

In order to minimize the number of equivalence classes we search for solutions that satisfy the negated formulas of the counter. $\neg f_2^3 = (1 \leftrightarrow (v_2 \leftrightarrow 1))$ is satisfied if $v_2 = 1$. Since $\neg f_1^3$ is contradictional to $\neg f_2^3$ the formula evaluates to 0 and $\neg f_0^3$ is 0, too. The don't care state q_2 has to be assigned to the set of final states and the minimal number of equivalence classes is $2^2 \cdot f_2^3 + 2^1 \cdot f_1^3 + 2^0 \cdot f_0^3 = 3$.

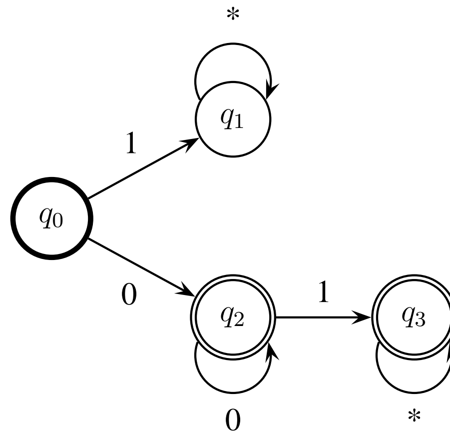

Figure 20: Restricted automaton after selection of final states

After selecting the don't care state to the set of final states state q_2 and q_3 belong to the same equivalence class and can be merged into a single state. The resulting minimized automaton which represents the Presburger formula $(x \geq 0)$ can be seen in figure 18.

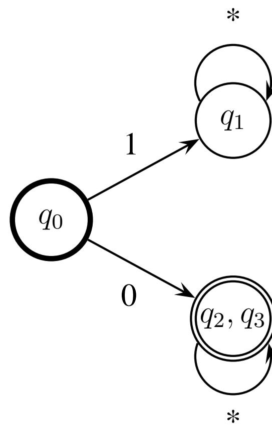

Figure 21: Restricted automaton after minimization

6 Experimental Results

To evaluate the approach, the algorithm was implemented and integrated in the symbolic model checker Beryl that is part of the Averest toolbox [20]. The application to some benchmarks was performed on a Intel Xeon processor with 3.06 GHz clock frequency and 2GB RAM. The representation and manipulation of Presburger formulas is done on the basis of deterministic finite automata (DFAs). Since a module for propositional logic is required the CUDD package is used [23].

The first experiment tested the restriction of automata with various parameters (Table 1). The first column of the parameters specify the used bitwidth, the second column shows the number of variables and the last two columns show the number of formulas and the number of assignments, respectively. The number of tested automata are shown in the sixth column. The runtimes of the program are given in seconds.

The size of an automaton is dependent on the chosen bitwidth and the number of variables that occur in the formula. The number of tested automaton depend on the number of variables and formulas. It can be seen that an increasing number of tested automata needs more time for computation. The automaton size has a large influence on the runtimes, the computation with three bits and three variables needed more than one day.

Benchmark	Parameters				Automata	Time
	bit	var	form	assign		
testpres	2	2	10	10	20	10.544
testpres	2	3	10	10	30	32.066
testpres	2	4	10	10	40	117.561
testpres	3	2	10	10	20	5.899
testpres	3	3	3	3	9	> 1d
testpres	4	2	10	10	20	19.335
testpres	4	2	20	20	40	6523.360
testpres	4	2	50	50	100	6402.760

Table 1: Runtimes of automata restriction

Most of the tested automata had a result with the same size. In the trivial case, there are no don't care states (the restricting automaton accepted all words) and the result is the original automaton itself. For other cases nothing can be said since a resulting automaton may have the same size but accepts another language. The minimal automaton has two states and is obtained either when the restricting automaton accepts an empty set of words or the intersection of both languages is empty.

The next table shows examples of automata that have a result with a smaller size. Column 1-5 show the number of states of the automaton that has to be restricted $\mathcal{A}_1$, the restricting one $\mathcal{A}_2$, their product $\mathcal{A}_1 \times \mathcal{A}_2$, the number of don't care states, and the size of the result $\mathcal{A}'$. DC/PA is the percentage share of don't care states within the product automaton and reachable is the size of the product automaton in relation to its maximal possible size given in percent $(|\mathcal{A}_1 \times \mathcal{A}_2| \cdot 100)/(|\mathcal{A}_1| \cdot |\mathcal{A}_2|)$. RelMin shows the percentage share of the difference between the size of $\mathcal{A}_1$ and $\mathcal{A}'$ computed by $(100 - (\mathcal{A}'/\mathcal{A}_1) \cdot 100)$.

$\|\mathcal{A}_1\|$	$\|\mathcal{A}_2\|$	$\|\mathcal{A}_1 \times \mathcal{A}_2\|$	DC States	$\|\mathcal{A}'\|$	DC/PA	Reachable	RelMin
3	4	4	2	2	50.0	33.3	33.3
4	7	7	5	2	71.4	33.3	50.0
5	8	23	19	4	82.6	57.5	20.0
7	14	22	20	2	90.9	22.4	71.4
4	4	5	3	2	60.0	31.3	50.0
4	4	6	4	2	66.6	37.5	50.0
6	6	15	12	5	80.0	41.6	16.6
4	3	4	1	3	25.0	33.3	25.0
9	6	22	18	7	81.8	40.7	22.2
17	14	44	40	14	90.0	18.4	17.6
21	10	45	41	10	91.1	21.4	52.4

Table 2: Results with smaller size

Tabular 3 shows some restricted automata that have the same size as the original automaton. Again the number of states from the different automata, the number of don't care states and the percentage share of the don't care states within the product automaton and the reachable states of $\mathcal{A}_1 \times \mathcal{A}_2$ in relation to the maximal possible size are given.

$\lvert\mathcal{A}_1\rvert$	$\lvert\mathcal{A}_2\rvert$	$\lvert\mathcal{A}_1 \times \mathcal{A}_2\rvert$	DC States	$\lvert\mathcal{A}'\rvert$	DC/PA	Reachable
2	4	4	1	2	25.0	50.0
3	5	6	1	5	16.6	40.0
7	9	49	6	7	12.2	77.7
15	19	93	50	15	53.7	32.6
6	6	20	8	6	40.0	55.5
7	7	37	12	7	32.4	75.5
5	3	6	1	5	16.6	40.0
7	5	25	12	7	48.0	71.4
14	7	40	13	14	32.5	40.8
14	13	103	89	14	86.4	56.5

Table 3: Results with equal size

The fact that the result is not smaller than the original automaton has two reasons. On the one hand the product automaton has a large number of reachable states that additionally must have a large difference to the size of $\mathcal{A}_1$ and $\mathcal{A}_2$ and on the other hand the percentage share of the don‘t care states is very low.

The second experiment checked specifications within a Kripke structure that modeled a program for finding the minimal element in an integer array with linear search. The program was tested with different numbers of array elements and least fixpoint computations. The runtimes for the specification with common image computation and image computation with the restrict operator are given in seconds.

Benchmark	Array	Fixpoint			
		AF	EF	$AF_{\downarrow}$	$EF_{\downarrow}$
LinSearch	4	0.01	0.01	2.44	2.79
LinSearch	6	0.02	0.03	37.36	29.99
LinSearch	8	0.03	0.03	40.75	59.34
LinSearch	10	0.11	0.18	766.68	493.8

The runtimes of the fixpoint computations with the common image computation are much faster than the one with the restrict operator since the minimization of automata requires a lot of runtime. In this experiment, the method could make use of non-trivial minimizations of the original automaton.

7 Conclusion

This work presents an exact technique to reduce the size of a finite automaton with respect to a second one. Automata can be translated from Presburger formulas and are efficient data structures for storing and manipulating possibly infinite sets during symbolic simulation. Since not all variable assignments of a formula are relevant we aimed to compute a minimal automaton that only may differ from the original automaton in the don't care set of assignments. Here, a method is proposed that restricts a finite automaton with respect to another one that represents the care set. It relies on the minimization of their product automaton by computing an equivalence relation and determining the smallest number of equivalence classes. Since the product automaton has next to the common state sets additionally states that have an unspecified behavior the minimal result depends on the assignment of the don't care states to the set of final or non-final states. This problem is reducable to a problem that is known to be NP-complete.

The procedure is safe such that the resulting automaton never has more states than the original automaton. Due to the definition of the restrict operator the minimized automaton is not uniquely defined, it may happen that there is more than one minimal automaton representing a different formula. According to the results we obtain with the algebraic properties of the restrict operator we cannot use the resulting automaton for an exact image computation. Nevertheless, it can be used to compute an overapproximation for least fixpoints and an underapproximation for greatest fixpoints.

Since the computation of a minimal automaton requires a lot of runtime the application of the restrict operator for fixpoint computations is rather slow. It would be advantageous to find heuristics that are able to reduce automata fast in order to obtain better runtimes.

References

[1] D. Angluin. Learning regular sets from queries and counter examples. *Information and Computation*, 75,no.2:87–106, 1987.

[2] A.W. Biermann and J.A. Feldmann. On the synthesis of finite state machines from samples of their behavior. *IEEE Transactions on Electronic Computers*, 21:529–597, 1972.

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT, London, England, 1999.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Studienreihe Informatik. Freeman, San Francisco, 1979.

[5] E.M. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[6] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. 14:350–359, June 1965.

[7] S. Gören and F.J. Ferguson. Chesmin: A heuristic for state reduction in incompletely specified finite state machines. In *IEEE/ACM Design, Automation and Test in Europe*, pages 248–254, Paris,France, March 2002.

[8] H. Higuchi and Y. Matsunaga. A fast state reduction algorithm for incompletely specified finite state machines. In *33rd Design Automation Conference*, pages 463–466, 1996.

[9] J.E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical Report 71/190, Stanford University, 1971.

[10] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[11] F. Somenzi J.-K. Rho, G.D. Hatchel and R.M. Jacoby. Exact and heuristic algorithms for the minimization of incompletely specified state machines. In *IEEE Transactions of Computer-Aided Design of Integrated Circuits and Systems,Vol.13,no.2*, pages 167–177, February 1994.

[12] L.N. Kannan and D. Sarma. Fast heuristic algorithms for finite state machines. In *Proc. of European Design Automation Conference*, pages 192–196, February 1991.

[13] J.H. Kukula, T.R. Shiple, and A. Aziz. Techniques for implicit state enumeration of EFSMs. In G. Gopalakrishnan and P.J. Windley, editors, *Conference on Formal Methods in Computer Aided Design (FMCAD)*, volume 1522 of *LNCS*, pages 469–482, Palo Alto, California, USA, 1998. Springer.

[14] M.C. Paull and S.H. Unger. Minimizing the number of states in incompletely specified sequential switching functions. *IRE Transactions on Electronic Computers*, EC-8:356–367, September 1959.

[15] J.M. Pena and A.L. Oliviera. A new algorithm for the reduction of incompletely specified finite state machines. In *International Conference on Computer-Aided Design*, pages 482–489, 1998.

[16] C. P. Pfleeger. State reduction in incompletely specified finite state machines. C-22:1099–1102, December 1973.

[17] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in we lchem die Addition als einzige Operation hervortritt. In F. Leja, editor, *Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich, Warszawa 1929 (Comptes–rendus du I Congrès des Mathématiciens des Pays Slaves, Varsovie 1929)*, pages 92–101 (supplement on p. 395), Warszawa, 1930.

[18] M. Presburger. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12:225–233, 1991. Transaltion and commentary by D. Jacquette.

[19] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[20] K. Schneider and T. Schuele. Averest: Specification, verification, and implementation of reactive systems. In *Conference on Application of Concurrency to System Design (ACSD)*, St. Malo, France, 2005. participant's proceedings.

[21] T. Schuele and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 153–162, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[22] F. Somenzi. Binary decision diagrams, 1999.

[23] F. Somenzi. CUDD: CU decision diagram package, release 2.4.1, 2005. http://vlsi.colorado.edu.

[24] R. Brayton T. Kam, T. Villa and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *31st ACM/IEEE Design Automation Conf.*, pages 684–690, June 1994.

[25] B. W. Watson. A taxonomy of finite automata minimization algorithms. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[26] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In S. Graf and M.I. Schwartzbach, editors, *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 1–19, Berlin, Germany, March 2000. Springer.

A Acknowledgement

I would like to thank my supervisor Dipl.-Inf. Tobias Schüle and Prof. Dr. rer. nat. Klaus Schneider very much for the friendly and helpful support while writing my diploma thesis.

Moreover, I won't forget that my parents took care of my daughter Sophie so that I had quiet times to reconsider my work. Finally, I am grateful to everyone who encouraged me and advanced my studies. Last, but not least: Thanks for your great patience with me, Sophie!