UNIVERSITEIT TWENTE.

MASTER'S THESIS

Symbolic Model Checking using Zero-suppressed Decision Diagrams

Author:
Maryam HAJIGHASEMI

Graduate committee:
Prof. Jaco VAN DE POL
Prof. Arend RENSINK
Tom VAN DIJK , MSc.

November 2014

Abstract

Formal Methods and Tools Group

Department or Computer Science

Master of Science

Symbolic Model Checking using Zero-suppressed Decision Diagrams

by Maryam HAJIGHASEMI

Symbolic model checking represents the set of states and transition relation as Boolean functions, using Binary Decision Diagrams (BDDs). One alternative to common BDDs are Zero-suppressed Decision Diagrams (ZDDs), which are BDDs based on a new reduction rule. The efficiency of ZDD representation, in comparison with the original BDD, is noticeable especially for sparse state spaces, in which the actual number of existing states is much smaller than the total number of possible states.

To the best of our knowledge, the current implementation for ZDDs is using fixed set of variables, i.e., domain for all possible diagrams. This may result in increase of size for each diagram. The main goal of this project is to develop an implementation of ZDDs with possibility of having different domains for specific diagrams. The secondary goal is to investigate the efficiency of ZDDs in comparison with BDDs, e.g. memory usage and running time, for reachability algorithm.

Contents

Abstract ii

Contents iii

List of Figures v

List of Tables vi

1 Introduction 1

2 Preliminaries 3
- 2.1 Model Checking . . . 3
  - 2.1.1 Reachability Algorithm . . . 4
- 2.2 Binary Decision Diagrams . . . 5
- 2.3 Zero-Suppressed Binary Decision Diagram . . . 7
- 2.4 CUDD . . . 10
- 2.5 Sylvan . . . 12

3 Implementation of ZDDs 14
- 3.1 Notations . . . 14
- 3.2 Converting BDD to ZDD . . . 15
- 3.3 `Extend` operation . . . 16
- 3.4 `ITE` operation . . . 17
- 3.5 `Not` operation . . . 20
- 3.6 `Exist` (∃) operation . . . 21
- 3.7 `Rename` operation . . . 22
- 3.8 `RelProd` operation . . . 24
- 3.9 `RelProdS` operation . . . 27

4 Experiments 29
- 4.1 Setups . . . 29
- 4.2 Sokobon models . . . 31
  - 4.2.1 Results . . . 32
- 4.3 BEEM models . . . 35
  - 4.3.1 Results . . . 36

5 Related work 41

6 Conclusions and Future Work **43**
6.1 Conclusions 43
6.2 Future Work 44

A Correctness Proofs **45**
A.1 `Extend` Operation 46
A.2 `ITE` Operation 47
A.3 `Not` Operation 52
A.4 `Exist` Operation 54
A.5 `Rename` Operation 57
A.6 `RelProd` Operation 59
A.7 `RelProdS` Operation 61

Bibliography **63**

List of Figures

2.1 Music player state graph 4
2.2 BDD representation of $F = (x_1 \vee x_2) \wedge x_3$ 6
2.3 Apply reduction rules on $F = (x_1 \wedge x_2) \vee x_2$ 7
2.4 ZDD *pD-deletion* rule 8
2.5 Simple ZDD examples 8
2.6 BDD and ZDD representation $F = \overline{x_1}x_2\overline{x'_2}$ on different domains 9
2.7 ZDD and BDD representation of F and F' 10
2.8 ZDD representation of $x_1\overline{x_2} \vee \overline{x_1}x_2$ with different domains Σ and Σ' 11
2.9 ZDD representation of $\exists\Sigma.(\overline{x_1}x_2\overline{x'_2})$ using different methods 12
2.10 Representing set $X = \{x_1, x_2, x_3\}$ in Sylvan 13

4.1 Transition relation size of Sokoban examples for ZDD and BDD 33
4.2 Size of reached states in different iteration for BEEM models 33
4.3 Size of transition relation groups for BEEM models 37
4.4 Correlation between the reduction of number of calls to `gc` and speedup, i.e., the decrease in computation time by using ZDD (s) 38
4.5 Speedup by percentage of reduction in number of times `gc` is called using ZDD 38

List of Tables

2.1 Mandatory operations for reachability algorithm 5

3.1 Calculating Boolean operations using `ITE` 18

4.1 Number of iterations for the reachability algorithm and number of nodes for BDD and ZDD representation of reachable states for Sokoban screens 31
4.2 Number of iterations and groups of transition relation for the reachability algorithm and number of nodes representing reachable states using BDD and ZDD for BEEM database models 31
4.3 Used memory for Sokoban models in number of used buckets 33
4.4 Computation time of Sokoban examples (ms) 34
4.5 Number of times `RelProdS` operation called 34
4.6 CPU profile of screen.387 using ZDD and BDD 35
4.7 Computation time of Sokoban examples using garbage collection (ms) 36
4.8 CPU profile of schedule_world.3.8 using ZDD and BDD 39
4.9 Computation time of BEEM models using BDD and ZDD 39
4.10 Number of function calls for `ITE` and `OR` operations using BDD and ZDD for BEEM models 40

Chapter 1

Introduction

Model checking is a formal verification technique used to verify whether a given model of a system satisfies certain desired properties. It is applied in areas like hardware verification and software engineering. Nowadays, model checking is used for realistic designs, with a large number of components. This leads to exponential growth of state space model of the system, which is called state explosion problem. Using Boolean formulas to represent sets and relations, rather than individual elements for each state, helps to avoid this problem. This method is called symbolic model checking [18].

Binary Decision Diagrams (BDDs) are used in symbolic model checking to represent Boolean formulas. Various existing packages have implemented the necessary operations to use BDDs, like BuDDy [17], CUDD [27], and Sylvan [29]. One alternative to common BDDs are Zero-suppressed Decision Diagram (ZDD) [22]. ZDD encompasses all the characteristics of BDD except that it benefits from a new reduction rule. This new reduction rule causes a noticeable improvement in the space consumption, in comparison to the original BDD. This happens specifically for sparse state spaces, i.e., when the number of states are much smaller than the number of possible states that may appear. Although ZDDs have been used in several areas, in the model checking applications, it has been used only for Petri-nets, since their state spaces are very sparse [31].

In this project we investigated how ZDDs could be exploited for symbolic model checking. One core challenge was that there is no existing complete package for this purpose. CUDD and EXTRA [24] are the only two packages that support ZDDs, but there are two problems with using these ZDD implementations for model checking. One is that the set of variables, i.e., domain, is fixed and same for all decision diagrams, which reduces the efficiency of ZDD. Another problem is that some required functions for reachability implementation are missing in CUDD, and are implemented in different way as expected in EXTRA, such as $\exists$. In Section 2.4, both problems are explained in detail.

So as the first step of this project we implemented a ZDD library that supports the needed operations for model checking, especially for the reachability algorithm, in Sylvan [29], a parallel BDD library. We chose Sylvan since it uses the BDD structure which is reusable for ZDDs. Moreover, addition of the domain attribute is easy to handle in it. Then we compared the performance of ZDDs and BDDs as two ways of representing sets of states, and transition relations.

We performed our experiments with several models of Sokoban puzzles and from the BEEM database [26], which is a database for explicit model checking, using our implemented ZDD package as an extension of Sylvan. We compared the results with the implementation of the same algorithm using the BDD operations of Sylvan. The results show that ZDDs are efficient on memory usage in the reachability algorithm. We also had speedup using ZDDs for some examples, but it/ did not occur for all cases.

Chapter 2 introduces BDDs and ZDDs. This chapter also explains the required operations for reachability. The ZDD algorithms and implementation of operations like `ITE`, `Not`, `Exist` and `Rename` are explained in chapter 3. Chapter 4 describes reachability analyses on models from the BEEM database and some Sokoban example. These experiments compare BDDs and ZDDs in both execution time and memory usage. Some ZDD applications in other areas are collected in chapter 5, and chapter 6 concludes the report and represents possible ideas for future work.

Chapter 2

Preliminaries

This chapter introduces the background knowledge of model checking and Binary Decision Diagram (BDD) in Sections 2.1 and 2.2. We also discuss ZDDs in details in Section 2.3, and limitations of CUDD for reachability algorithm in Section 2.4.

2.1 Model Checking

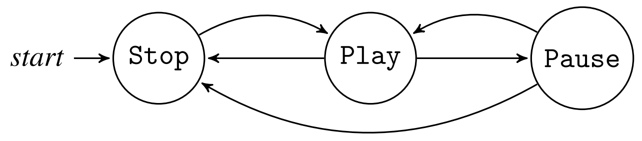
Model checking is a technique for verifying specific properties of a system. The purpose is to check whether given properties hold for a given model of a system. For example, if a system suffers from a deadlock or if it meets a safety requirement, or if there is a possibility of reaching a specific state in the state graph.

A *model* describes all possible behaviors of a system. Many systems can be modeled as state graphs, which can be defined as a tuple (S, T, I, Σ) where S is a set of states, T is a transition relation, $I \subseteq S$ is a set of initial states, and Σ is the set of variables, i.e., domain.

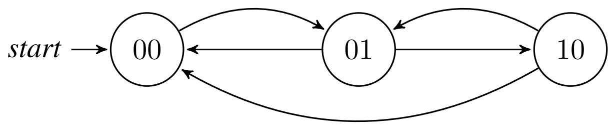
Each state in S is a valuation of variables in Σ. Let $\Sigma = \{x_1, x_2\}$ in which x_1 and x_2 are Boolean variables, then for instance, $\overline{x_1}x_2$ represents a state, where x_1 is $False$ and x_2 is $True$. We can define a subset of all possible states by using Boolean function F. For instance, $F = x_1$ represents the set of states in which x_1 is $True$.

A transition relation T, is a binary relation, $T \subseteq S \times S$, for which we use Boolean functions as representation. Let s, s' be a vector of variables in X, then $T(s, s')$ represents transitions from the set of states s to the set of states s'. For example, $T(s, s') = \overline{x_1}x'_1x'_2$ shows there are two transitions from states $\{\overline{x_1}x_2, \overline{x_1}\,\overline{x_2}\}$ to x_1x_2.

Example 2.1. *Consider a simple music player with three operations, represented by a set of states* $\{\texttt{Play}, \texttt{Pause}, \texttt{Stop}\}$. *We start with the instrument being stopped,* `Stop` *state. It is not* *possible to* `Pause` *when the music is stopped. The following state graph models this music player using 3 states and 5 transition relations.*



We use Boolean variables to represent states and transitions, i.e., we assign each state with a boolean string:



Now by using two Boolean variables x_1, x_2, we can easily show each state as follows:

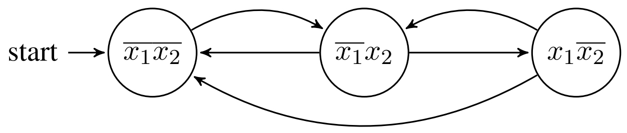


FIGURE 2.1: Music player state graph

Model checking can be divided in two categories: explicit-state, and symbolic model checking. The former is being done by enumerating and storing all states individually, whereas the latter represents the set of states, and transition relations as Boolean functions. In this report we use symbolic model checking.

2.1.1 Reachability Algorithm

Reachability analysis is one of the main processes of model checking. The goal is to find all reachable states from an initial set of states I with transition relation T. We can use the set of reachable states to verify whether certain properties hold or not. State s is a reachable state, if there is a path from one of the states in I to s, according to a given transition relation T. To calculate all reachable states, starting from initial states we find the next reachable states using transitions, the process continues until no new reachable state is found. Since we are assume that the state space in finite, this process is guaranteed to terminate.

In Example2.1, the initial state is $\overline{x_1}\overline{x_2}$, and in the first iteration, state $\overline{x_1}x_2$ is reachable. In the second iteration, state $x_1\overline{x_2}$ is also reachable. Since in the third iteration we have the same set of reachable states, the algorithm terminates. The *reachability* Algorithm 1 is as follows:

Algorithm 1 Reachability algorithm

```
1: function REACHABILITY($I$,$T$,$\Sigma$,$\Sigma'$)
2:                      ▷ $I$: initial state, $T$: transition relations, variables in $\Sigma'$ renamed with $\Sigma$
3:     $states, new \leftarrow I$
4:     while $new \neq \emptyset$ do
5:         $new \leftarrow \exists\Sigma.(new \wedge T)[\Sigma' \setminus \Sigma]$     ▷ calculate reachable state in the next iteration
6:         $states \leftarrow states \vee new$                ▷ add new reachable states
7:     return $states$
```

In this algorithm we find new reachable states in line 7. First $new \wedge T$ finds the possible transitions from reached states in the last iteration. Then we abstract the set of variables in Σ, using $\exists\Sigma$ (it is also known as `Exist` in this report), that results in the next reachable states in domain Σ'. All variables in Σ' are substituted by variables in Σ, using `Rename` operation, to have reachable states in the next iteration. In line 8, these new reached states are added to previous ones. Table 2.1 shows the required operations for reachability algorithm and the corresponding line that is used in the algorithm.

	Operation name	used in line
1	`Union`	8
2	`Intersect`	7
3	`Exist`($\exists$)	7
4	`Rename`	7

TABLE 2.1: Mandatory operations for reachability algorithm

2.2 Binary Decision Diagrams

Binary Decision Diagrams (BDD), were firstly proposed by Akers in [3] and later developed by Bryant [7]. A BDD is a graph for representing Boolean functions with restriction on the ordering of variables in the graph. It can be used to store sets of states in symbolic model checking. A Shannon decomposition of a Boolean function, as defined below, can be represented by a BDD, which is a directed, acyclic graph.

Shannon decomposition and cofactor: Let F be a boolean function on $\Sigma = \{x_1, x_2, \ldots, x_n\}$. The following identity is Shannon decomposition of F with respect to x_i:

$$F = (x_i \wedge F_{x_i=1}) \vee (\overline{x_i} \wedge F_{x_i=0})$$

where $F_{x_i=1}$ and $F_{x_i=0}$ are F with the argument x_i equal to 1, and 0, respectively. Which is also defined as follows:

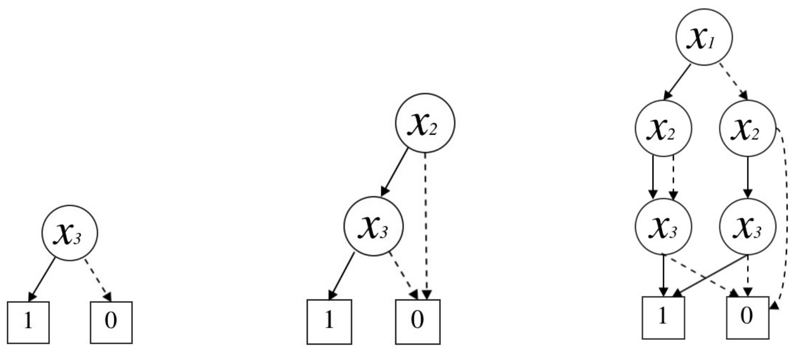
$$F_{x_i=v}(x_1,\dots,x_{i-1},x_i,\dots,x_n) = F(x_1,\dots,x_{i-1},v,\dots,x_n)$$

A BDD has two types of nodes, terminal and non-terminal. A terminal node represents a constant value of 0 or 1, it has no outgoing edges. A non-terminal node represents an input variable index, and it has two outgoing edges labeled 0 and 1. The one labeled 0 (0-edge) points to the sub-graph $F_{x=0}$, and other one (1-edge) points to the sub-graph $F_{x=1}$.

In this report, we use rectangles as terminal nodes with 0 or 1 labels, and non-terminal nodes are represented by circles containing the variable index. A dashed edge indicates a 0-edge and solid edge indicates a 1-edge.

An Ordered BDD is a BDD where there is a total ordering $\prec$ over the set of variables. Which means if $x_i \prec x_j$, then all nodes with x_i precede all nodes with x_j.

Figure 2.2 shows the step by step BDD representation of the Boolean function $F = (x_1 \vee x_2) \wedge x_3$. The variable ordering in this graph is $x_1 \prec x_2 \prec x_3$. According to the ordering we start from x_1, and we have $F_{x_1=1} = x_3$ (Figure 2.2(a)) and $F_{x_1=0} = x_2 \wedge x_3$ (Figure 2.2(b)). The result of applying the Shannon decomposition is $F = (x_1 \wedge x_3) \vee (\overline{x_1} \wedge x_2 \wedge x_3)$. To complete the representation of F by a BDD, the mentioned procedure should be repeated for x_2 and x_3. The final BDD is given in Figure 2.2(c).



(a) x_3 (b) $x_2 \wedge x_3$ (c) $(x_1 \vee x_2) \wedge x_3$

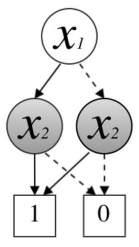
FIGURE 2.2: BDD representation of $F = (x_1 \vee x_2) \wedge x_3$

An ordered BDD is reduced if it satisfies two conditions: it should not contain any *redundant* nodes and it should not include any duplicate sub-graphs. A node in a BDD is called *redundant* node if it has two identical children. If the two mentioned conditions holds in a BDD it is called *Reduced Ordered BDD* (ROBDD). For instance, Figure 2.2(c) is an ordered BDD but not a ROBDD, since x_2 is a redundant node. In order to reduce a BDD, two rules should be applied:

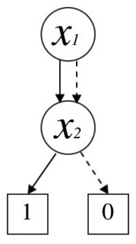
1. *S-deletion rule*: All redundant nodes must be deleted.

2. *Merging rule*: All duplicate sub-graphs must be deleted by sharing the sub-graphs among upper nodes.

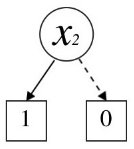
Figure 2.3, illustrates how to use these rules to reduce a BDD. All three BDDs represent the same Boolean function $F = (x_1 \wedge x_2) \vee x_2$. The colored nodes in Figure 2.3(a) are duplicated sub-graphs, which are eliminated by applying the *merging rule* in Figure 2.3(b). In the new generated BDD x_1 is a redundant node, and should be eliminated. Figure 2.3(c) represent an ROBDD, where both redundant node and duplicated sub-graphs are deleted.



(a) Node sharing



(b) Node deletion



(c)

FIGURE 2.3: Apply reduction rules on $F = (x_1 \wedge x_2) \vee x_2$

Applying reduction rules on an Ordered BDD guarantees a unique representation for an arbitrary given function. Therefore, a Reduced Ordered BDD provides us a canonical representation of Boolean functions. In this thesis we assume all BDDs are ROBDDs.

2.3 Zero-Suppressed Binary Decision Diagram

Zero-suppressed binary Decision Diagrams (ZDD) have been introduced by Minato in [19]. A ZDD is a BDD with a different deletion rule which is based on positive Davio expansion. Although this expansion forms the basic idea behind reduction rule in ZDD, however, ZDDs are constructed based on Shannon decomposition.

Positive Davio expansion: Let F be a boolean function on $\Sigma = \{x_1, \ldots, x_n\}$. The following identity is the positive Davio expansion of F with respect to x_i, where $x \oplus y = (x \wedge \overline{y}) \vee (\overline{x} \wedge y)$:

$$F = F_{x_i=0} \oplus x_i(F_{x_i=0} \oplus F_{x_i=1})$$

ZDDs reduce using *pD-deletion rule* [14], which is explained as below.

pD-deletion rule: A node x should be deleted, if its 1-edge points to a 0-terminal, and its 0-edge points to a node $F_{x=0}$. Since, by positive Davio decomposition rule we have $F = F_{x=0}$, all edges leading to x should be redirected to the node $F_{x=0}$. This process is shown in Figure 2.4.

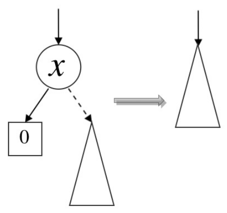
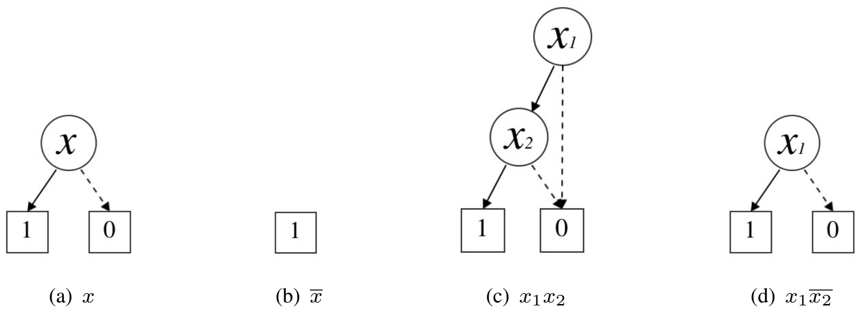


FIGURE 2.4: ZDD *pD-deletion* rule

This deletion rule is asymmetric with respect to 0-edge and 1-edge of a node. In the other word, we do not eliminate nodes whose 0-edge points to a 0-terminal. Note that *S-deletion rule* is not being used here any more, so nodes whose two edges point to the same node must be kept in the diagram. Examples of simple ZDDs are given in Figure 2.5. For instance, in Figure 2.5(c) the absence of variable x_2 for negative evaluation of x_1 is because the 1-edge of x_2 points to the 0-terminal.



(a) x (b) $\overline{x}$ (c) x_1x_2 (d) $x_1\overline{x_2}$

FIGURE 2.5: Simple ZDD examples

Same as BDDs, to have a unique representation of ZDDs, the variable ordering should also be fixed, since using different ordering simply changes the decision diagram. For a ZDD, input domain should also be fixed, otherwise it can be considered as a representation of different functions. If a variable doesn't appear in a Boolean formula, it can be both 0 and 1. This means the corresponding node is redundant in decision diagram. Since redundant nodes are eliminated in BDDs, adding new variables to domain does not affect the canonical representation of a function. However, since in ZDDs we don't eliminate redundant nodes, therefore the domain should be fixed. The following theorem ensure the uniqueness of ZDD.

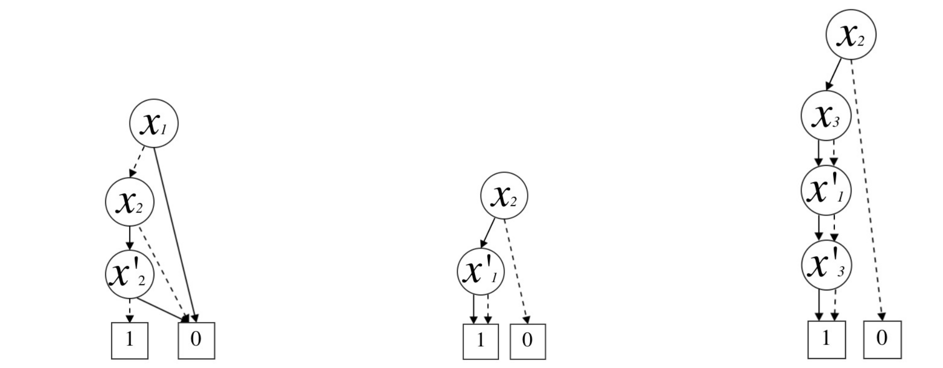
Theorem 2.1. *ZDD can uniquely represent a Boolean function if the variable domain and ordering are fixed[22].*

In a path of a Decision Diagram variables are divided into three categories:

- Positive: variables with value 1.
- Negative: variables with value 0.
- don't care: variables with both 0 and 1 value.

In a BDD, reduced variables in a path from root to a terminal node, are *don't care* variables. This means that the related node is redundant, and deleted because of S-deletion rule. In a ZDD, variables that are skipped in a path from root to a terminal node, has negative value, and deleted based on *pD-deletion* rule.

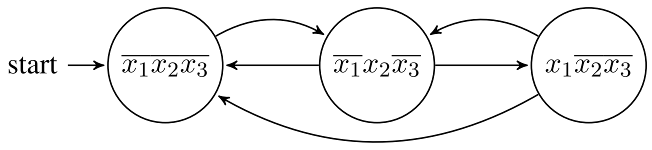
In the music player example, outgoing transitions from `Play` state are $F = \overline{x_1}x_2\overline{x'_2}(\overline{x'_1} \vee x'_1) = \overline{x_1}x_2\overline{x'_2}$. Figures 2.6(a) and 2.6(b) show BDD and ZDD representation of these transitions on $\Sigma = \{x_1, x_2, x'_1, x'_2\}$, respectively. In this example x_1 and x'_2 are negative, x_2 is positive and x'_1 is don't care. In case of using different domains to represent the same function F, like $\Sigma' = \{x_1, x_2, x_3, x'_1, x'_2, x'_3\}$, then x_3 and x'_3 are also don't care. So the same BDD still represents F, but the ZDD representation is different as shown in Figure 2.6(c). As a result, a fixed domain is necessary to represent a Boolean formula uniquely by ZDDs.



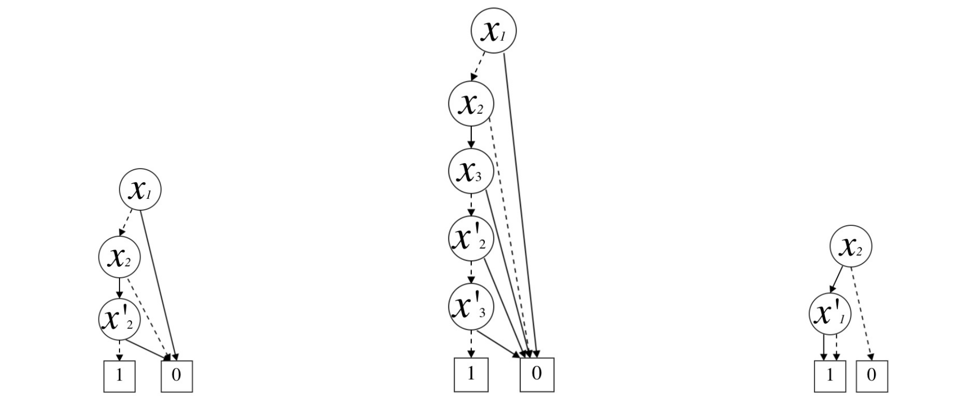
(a) BDD on Σ and Σ' domain (b) ZDD on Σ domain (c) ZDD on Σ' domain

FIGURE 2.6: BDD and ZDD representation $F = \overline{x_1}x_2\overline{x'_2}$ on different domains

The main advantage of ZDDs is that it is more efficient for sparse state space comparing to BDDs [22]. Which means the number of states are much smaller than the number of possible states that may appear. In the other words, most of the variables are assigned to zero in the Boolean formula. For instance, back to our music player example with the outgoing transitions from `Play` on $\Sigma' = \{x_1, x_2, x_3, x'_1, x'_2, x'_3\}$ domain. The music player can be also abstracted as follows. Then the transition is $F' = \overline{x_1}x_2\overline{x_3}\,\overline{x'_2x'_3}$. As we can see in Figure 2.7, same ZDD represent both F and F', since F' had two more negative node than F which are suppressed in ZDD. However, the BDD representation of F' has 5 nodes while only 2 nodes need to represent it by ZDD.



In this simple example, there are two solutions that both ZDDs and BDDs can represent the same function in different ways, and one of them become more efficient. But there are many cases that are more complex and sparse. In these cases ZDDs may be more efficient than BDDs, in both memory usage and computation time. Example of it can be found in chapter 4.



(a) BDD for $F = x_1\overline{x_2}\overline{x'_2}$ (b) BDD for $F' = x_1\overline{x_2}\overline{x_3}\overline{x'_2x'_3}$ (c) ZDD for both$F$ and F'

FIGURE 2.7: ZDD and BDD representation of F and F'

2.4 CUDD

CUDD[27] is a package supporting three types of decision diagram: BDD, ADD [4] and ZDD. It is one of the well-known packages for BDD, and it has all basic functions that are needed to use BDDs for model checking, while it has limited functions for ZDDs. There are couple of ZDD procedures in the CUDD package that covers the basic operations for ZDDs, such as Union, Intersect, and If Then Else(ITE). As mentioned in table 2.1, implementing the reachability algorithm needs some additional operations, namely, $\exists$, which remove some variables from a DD, and `Rename` that substitutes a variable with another one in DD. These two operations are not supported in CUDD.

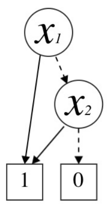
EXTRA[24] library is an extension of CUDD package. It uses the same structure as CUDD and adds some of the missing functions in CUDD like $\exists$ and `Rename`. There is a list of ZDD procedures that EXTRA adds to CUDD in [25]. So all the mandatory functions for reachability algorithm are supported by EXTRA .

But there are still two problems that prevent us from using the ZDD implementation of EXTRA, for symbolic model checking: (i) The domain attribute is fixed for all defined decision diagrams. (ii) The $\exists$ operation result is not as expected for relational product implementation, since the domain does not change properly that is a consequence of the first problem.

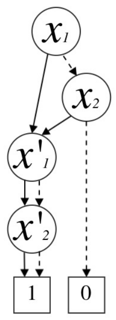
As described in Section 2.3, ZDD representation of a set of states, requires having a specified domain of variables, while for BDDs it is not necessary. In CUDD same domain of variables is considered for all ZDDs, that includes all defined variables. So the domain includes all the variables from 0 to the largest defined variable in the implementation. For example, if the initialized number of ZDD variables is 5 then the domain Σ is $\{x_0, x_1, x_2, x_3, x_4, x_5\}$. This property limits the selection of domain variables. For instance, it is not possible to set the domain to be a set of odd numbers or ranging between 5 to 10, instead of all possible values for variables. This will cause the generation of large diagrams, and hence decreases the efficiency of ZDDs.

As we have seen before, if the state space of a model is represented by $\Sigma = \{x_1, \ldots, x_n\}$, then the related transition relations represents using twice variable in Σ including both x_i and x'_i for each variable, that represent the current and next value of each variable, respectively.

While in CUDD, all variables are included in both cases, where half of them are don't care variables for state space representation. In the music player example 2.1, being in `Play` or `Pause` state formulates as $F = x_1\overline{x_2} \vee \overline{x_1}x_2$, where the domain can be either $\Sigma = \{x_1, x_2\}$ or $\Sigma' = \{x_1, x_2, x'_1, x'_2\}$. The following figures show representation of same function using two different domains.



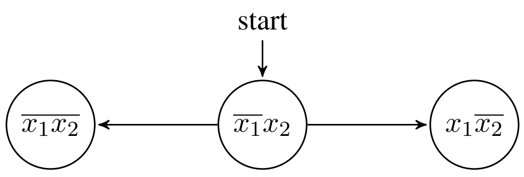
(a) $\Sigma = \{x_1, x_2\}$



(b) $\Sigma' = \{x_1, x_2, x'_1, x'_2\}$

FIGURE 2.8: ZDD representation of $x_1\overline{x_2} \vee \overline{x_1}x_2$ with different domains Σ and Σ'

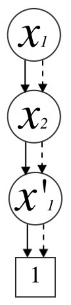
The second problem relates to the implementation of ZDD operations in CUDD. Consider the following example for reachability algorithm, that whether the `Pause` state is reachable from `Play` state in music player Example 2.1. The following state diagram represents the simplified version of the example,where the initial state is $I = \overline{x_1}x_2$ and the outgoing transitions from this state are $T = \overline{x_1}x_2\overline{x'_2}$.



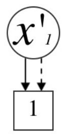
Reachable states from `Play` calculate in three steps based on Algorithm 1. First step is finding possible transitions from initial states using "$I \wedge T$", which is equal to T in this case. Next step is abstracting current state variables domain, which is $\Sigma = \{x_1, x_2\}$, using $\exists$ function (Chapter 3.6). The last step is renaming x'_1 to x_1 and x'_2 to x_2. The second step is calculated as follows.

$$\exists\Sigma.(T) = \exists\Sigma.(\overline{x_1}x_2\overline{x'_2}) = \overline{x'_2}$$

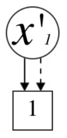
According to the definition of $\exists X.\mathcal{Z}_\Sigma(f)$, where $\mathcal{Z}_\Sigma(f)$ is the ZDD representation of f with domain variables Σ, variables in X remove from Boolean formula f. And the resulting domain can be either the same as the input domain (Σ), or excluding abstracted variables ($\Sigma - X$). The BDD representations of both methods are the same. For the first method, abstracted variables consider as don't care variables, that reduce in BDD. In the other method, these variables are not part of the result domain, so again are not present in diagram (see Chapter 2.3). But the ZDD representations are different, and are shown in Figure 2.9(a) and 2.9(b). The problem with



(a) using same domain as input $\Sigma = \{x_1, x_2, x'_1, x'_2\}$



(b) remove abstracted variables from domain $\Sigma' = \{x'_1, x'_2\}$



(c) using EXTRA implementation $\Sigma = \{x_1, x_2, x'_1, x'_2\}$

FIGURE 2.9: ZDD representation of $\exists\Sigma.(\overline{x_1}x_2\overline{x'_2})$ using different methods

EXTRA library is that, the domain is fixed but the result is the same as removing the variables from the domain. For example, the result of $\exists\Sigma.(\overline{x_1}x_2\overline{x'_2})$, is as shown in Figure 2.9(c), using EXTRA implementation of $\exists$. But as mentioned earlier the domain should be the same as input domain. This makes wrong interpretation of the result. As the variables are parts of the domain and are not presented in the diagram, they are considered as negative variables instead of don't cares. Because of these two problems we decided to develop a new implementation of ZDDs without the mentioned problems for existing implementation.

2.5 Sylvan

Sylvan is a parallel BDD package implemented using lock-less data structures and work-stealing [30]. In Sylvan, BDD operations include recursive tasks, which can be done in parallel. Most

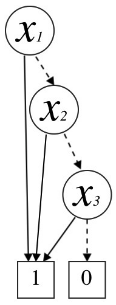


FIGURE 2.10: Representing set $X = \{x_1, x_2, x_3\}$ in Sylvan

of the time for each node representing e.g. x_i, in the BDD, the operation is recursively called over the two sub-graphs, namely for $F_{x_i=0}$ and $F_{x_i=1}$. In Sylvan, these two function calls are calculated in parallel.

BDD operations include three memory operations, which make a massive use of memory. These three operations are cache lookup, cache store, and hash table lookup. The results of BDD operations are stored in an operation cache, which is a hash table that overwrites new data on old one in case of collision. Moreover, the BDD nodes are also stored in a unique table, which is a hash table with garbage collection support.

Sylvan also supports a set of variables and its related operations. This set of variables can be used as a domain as well. For example, the set $X = \{x_1, x_2, \cdots, x_n\}$ is represented by the Boolean function $F = x_1 \vee x_2 \vee \cdots \vee x_n$, which is stored using BDDs. Figure 2.10 is an example of a set representation in Sylvan. We also use the same set as representation for the ZDD domain in our implementation.

Chapter 3

Implementation of ZDDs

This chapter discusses the implementation of important ZDD operations for model checking. First, the used notations in the algorithms are represented. Then we present how to convert BDDs to ZDDs and vice versa. Next section is about `ITE` algorithm, which can be used to calculate Boolean operations such as `And` and `Or`. The `Rename`, `Exist` and `RelProd` algorithms are explained in next sections. `Rename` operation substitutes some Boolean variables of a ZDD with new variables. `Exist` operation calculates the ZDD representation of a Boolean function after abstracting a set of variables from it.

3.1 Notations

In this section we introduce the notations that we use in definitions and correctness proofs. Assuming that f is a Boolean formula with Boolean variables $\Sigma = \{x_1, x_2, ...\}$, $\mathbb{V}$ a set of Boolean variables and $\mathbb{Z}_\Sigma$ set of ZDDs with Σ domain, then BDD, ZDD, the top node of a ZDD, and empty domain are represented as follows:

$\mathcal{B}_\Sigma(f)$	BDD for boolean formula f under Σ domain
$\mathcal{Z}_\Sigma(f)$	ZDD for boolean formula f under Σ domain
$A.t$	the top node (lowest level) in A as a ZDD
$\emptyset$	the empty domain

In ZDD implementations, the method `CreateNode(CN)` is used to create or reuse ZDD nodes, which has the specification

$$\texttt{CN}(x, \mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g)) = \mathcal{Z}_{\Sigma\cup x}((x \wedge f) \vee (\overline{x} \wedge g))$$

This method creates a ZDD, which x is its top node, and two edges that point to g and f, expect when $g = 0$ it immediately return f as the result. f is the same as $F_{x=1}$, and g is $F_{x=0}$ regarding to Shannon decomposition. We will assume the notation $< x, \mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g) >$ as shorthand for $\texttt{CN}(x, \mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g))$.

Most BDD and ZDD operations are recursive using Shannon decomposition . These operations use sub-problems where a selected variable x, is assigned to 0 or 1, and calculate the result recursively based on these sub-problems.

In the following, it is defined how a ZDD is created recursively based on a Boolean formula, using the explained reduction rules in chapter 2.

Definition 3.1. Let f be a Boolean formula and $f_{x=v}$ use for Boolean formula f, where variable x is assigned to v, and $v \in \{1, 0\}$. Then $\mathcal{Z}_\Sigma(f)$ definition is as follows:

$$\mathcal{Z}_\emptyset(f) = \begin{cases} 1 & \text{if } f = 1 \\ 0 & \text{if } f = 0 \end{cases}$$

$$\mathcal{Z}_{(x,\Sigma)}(f) = \begin{cases} \mathcal{Z}_\Sigma(f_{x=0}) & \text{if } \mathcal{Z}_\Sigma(f_{x=1}) = 0 \\ < x, \mathcal{Z}_\Sigma(f_{x=1}), \mathcal{Z}_\Sigma(f_{x=0}) > & \text{Otherwise} \end{cases}$$

Definition 3.2 explains that the Boolean function that represented by the ZDD under domain Σ can be obtained by $\mathcal{Z}_\Sigma^{-1}$ operation.

Definition 3.2. Let x be the smallest variable (the lowest level) in the domain Σ. Then $\mathcal{Z}_\Sigma^{-1}$ is defined as follows, where v_Σ, $v \in \{0, 1\}$ is the ZDD representation of v with the domain Σ:

$$\mathcal{Z}_\Sigma^{-1}(A) = \begin{cases} 1 & \text{if } A = 1_\Sigma \\ 0 & \text{if } A = 0_\Sigma \end{cases}$$

$$\mathcal{Z}_\Sigma^{-1}(A) = \begin{cases} (x \wedge \mathcal{Z}_{\Sigma'}^{-1}(A_{x=1})) \vee (\overline{x} \wedge \mathcal{Z}_{\Sigma'}^{-1}(A_{x=0})) & x = A.t \\ \overline{x} \wedge \mathcal{Z}_{\Sigma'}^{-1}(A) & x < A.t \end{cases}$$

3.2 Converting BDD to ZDD

One method for verification of a ZDD operation is converting a BDD to ZDD, do the same operation on both of them. Then convert the result back to BDD and check if they are equal. These conversion can be calculated using `BDD-To-ZDD` (`B2Z`) and `ZDD-To-BDD` (`Z2B`) operations, which have the types and specifications

$$\texttt{B2Z} : \mathbb{B}_\Sigma{}^\circ \to \mathbb{Z}_\Sigma$$
$$\texttt{B2Z}(\mathcal{B}_\Sigma(f)) = \mathcal{Z}_\Sigma(f)$$

$$\texttt{Z2B} : \mathbb{Z}_{\Sigma}{}^{\circ} \rightarrow \mathbb{B}_{\Sigma}$$
$$\texttt{Z2B}(\mathcal{Z}_{\Sigma}(f)) = \mathcal{B}_{\Sigma}(f)$$

Definition 3.3. Let x be the top variable in Σ and let $A = \mathcal{B}_{(x,\Sigma')}(f)$. Then B2Z is defined as follows:

$$\texttt{B2Z}(A) = \begin{cases} A & \text{if } \Sigma = \emptyset \\ < x, \texttt{B2Z}(A, \Sigma'), \texttt{B2Z}(A, \Sigma') > & \text{if } A = 0 \text{ Or } A = 1 \text{ Or } x < A.t \\ < x, \texttt{B2Z}(A_{x=1}, \Sigma'), \texttt{B2Z}(A_{x=0}, \Sigma') > & \text{if } x = A.t \end{cases}$$

In this operation if the domain is empty, it means that BDD is only a terminal node, which is the same in ZDD. But variables that does not show up in a BDD, are removed by $S - rule$ (redundant rule). These variables do not eliminate in ZDD, thus we should create them as nodes that both edges point to a same sub-graph. In the other cases we create the nodes recursively based on Shannon decomposition, and if the 1-edge points to 0 the CN operation will eliminate this node, using $pD - deletionrule$.

Definition 3.4. Let x be the top variable in Σ and let $A = \mathcal{B}_{(x,\Sigma')}(f)$. Assume BCN be the same as CN operation, but creates a node based on BDD reduction rules. Then Z2B is defined as follows:

$$\texttt{Z2B}(A) = \begin{cases} A & \text{if } \Sigma = \emptyset \\ \texttt{BCN}(x, 0, \texttt{Z2B}(A, \Sigma')) & \text{if } A = 0 \vee A = 1 \vee x < A.t \\ \texttt{BCN}(x, \texttt{Z2B}(A_{x=1}, \Sigma'), \texttt{Z2B}(A_{x=0}, \Sigma')) & \text{if } x = A.t \end{cases}$$

In conversion of ZDDs to BDDs, if the level k is skipped in a path in ZDD, then there is a node p at level k that its 1-edge points to 0 in the equivalent BDD. Otherwise the same node is created in BDD, except the domain is empty.

3.3 `Extend` operation

In some cases it is needed to change the domain of a ZDD without changing the Boolean function, to match the conditions as an input of an operation like ITE, which all its input should have the same domain (see section 3.4). Extending the domain does not affect the BDD representation, since these new variables are assumed to be don't care. However, extension of the domain of a ZDD changes its representation. All added variables to domain should also add to the ZDD. It is not efficient to use more nodes to represent the same Boolean function, but sometimes it is needed to match the requirements of other operations. Extend(EXT) operation has type and specification

$$\texttt{EXT} : \mathbb{Z}_{\Sigma}, (x : \mathbb{V}) \rightarrow \mathbb{Z}_{\Sigma \cup x}$$
$$\texttt{EXT}(\mathcal{Z}_{\Sigma}(f), x) = \mathcal{Z}_{\Sigma \cup x}(f)$$

Definition 3.5. Let $A = \mathcal{Z}_{\Sigma}(f)$, and let y be the top variable of A. Then `EXT` is defined as follows:

$$\texttt{EXT}(A, x) = \begin{cases} A & x = y \\ < x, A, A > & x < y \\ < y, \texttt{EXT}(A_{y=1}, x), \texttt{EXT}(A_{y=0}, x) > & x > y \end{cases}$$

Theorem 3.6. *The result of* $\texttt{EXT}(\mathcal{Z}_{\Sigma}(f), x)$ *is equal to* $\mathcal{Z}_{\Sigma \cup x}(f)$.

(Proof): See Appendix A.1.

Theorem 3.7. *The result of* $\texttt{EXT}(A, x)$ *is a reduced ordered ZDD, if* A *is a reduced ordered ZDD.*

Proof. For case 1 it is true, since A is reduced ordered ZDD. Result in case 2 is ordered ZDD if x is less than all variables in A. According to the condition of case 2, x is less than the top node of A, and A is ordered so x is less than all variables in A. In case 3, the result is only a reduced ordered ZDD, if y is less than all variables in $\texttt{EXT}(A_{y=v}, x)$, and if $\texttt{EXT}(A_{y=v}, x)$ is a reduced ordered ZDD. $A_{y=v}$ can only have variables greater than y which are in Σ', since it is represented by an ordered ZDD and y is the top node of A. As A is a reduced ordered ZDD, $A_{y=v}$ is also a reduced ordered ZDD. So y is less than all variables in $\texttt{EXT}(A_{y=v}, x)$.

□

3.4 `ITE` operation

If-Then-Else (`ITE`) operation is one of the basic Boolean operations. Other Boolean operators such as $\wedge$, $\vee$ and $\oplus$ can also be calculated using this operation, which has the type and specification

$$\texttt{ITE} : \mathbb{Z}_{\Sigma}, \mathbb{Z}_{\Sigma}, \mathbb{Z}_{\Sigma} \rightarrow \mathbb{Z}_{\Sigma}$$
$$\texttt{ITE}(\mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma}(g), \mathcal{Z}_{\Sigma}(h)) = \mathcal{Z}_{\Sigma}\left((f \vee g) \wedge (\overline{f} \vee h)\right)$$

The result of this operation, is based on three ZDDs: A, B and C. If A is true then the result will be equal to B, else it is equal to C. We assumed that the domain of these three ZDDs are the same, otherwise the calculation would be more complex. The following table shows some of Boolean operations calculation using `ITE`.

Boolean Operation	`ITE` equivalence
$f \wedge g$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g), 0_\Sigma)$
$f \vee g$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), 1_\Sigma, \mathcal{Z}_\Sigma(g))$
$\overline{f} \wedge g$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), 0_\Sigma, \mathcal{Z}_\Sigma(g))$
$\overline{f \wedge g}$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(\overline{g}), 1_\Sigma)$
$\overline{f \vee g}$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), 0_\Sigma, \mathcal{Z}_\Sigma(\overline{g}))$
$f \oplus g$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(\overline{g}), \mathcal{Z}_\Sigma(g))$
$f \rightarrow g$	$\texttt{ITE}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g), 1_\Sigma)$

TABLE 3.1: Calculating Boolean operations using `ITE`

Definition 3.8. Assume $A = \mathcal{Z}_\Sigma(f), B = \mathcal{Z}_\Sigma(g), C = \mathcal{Z}_\Sigma(h)$, which are well-formed ZDDs. Let $A_{x=v}$ be shorthand for $\mathcal{Z}_{\Sigma'}(f_{x=v})$ and $\texttt{ITE}_{x=v}$ be shorthand for $\texttt{ITE}(A_{x=v}, B_{x=v}, C_{x=v})$ with $v \in \{0, 1\}$. Also let x be top variable in Σ. Then `ITE` is defined as follows:

$$\texttt{ITE}(A,B,C) = \begin{cases} C & A = 0_\Sigma \\ B & A = 1_\Sigma \\ B & B = C \\ A & B = 1_\Sigma \wedge C = 0_\Sigma \\ \texttt{Not}(A) & B = 0_\Sigma \wedge C = 1_\Sigma \\ < x, 0_{\Sigma'}, \texttt{ITE}_{x=0} > & (B.t \neq x \wedge C.t \neq x) \vee \\ & (A.t \neq x \wedge B.t = x \wedge C.t \neq x) \\ < x, C_{x=1}, \texttt{ITE}_{x=0} > & A.t \neq x \wedge C.t = x \\ < x, \texttt{ITE}(A_{x=1}, B_{x=1}, 0'_\Sigma), \texttt{ITE}_{x=0} > & A.t = x \wedge B.t = x \wedge C.t \neq x \\ < x, \texttt{ITE}(A_{x=1}, 0'_\Sigma, C_{x=1}), \texttt{ITE}_{x=0} > & A.t = x \wedge B.t \neq x \wedge C.t = x \\ < x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0} > & A.t = x \wedge B.t = x \wedge C.t = x \end{cases}$$

Note. Note that in the above definition all cases except the terminal cases are the simplified version of

$$< x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0} >$$

Theorem 3.9. *The result of* $\texttt{ITE}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(g), \mathcal{Z}_\Sigma(h))$ *is equal to* $\mathcal{Z}_\Sigma\big((f \vee g) \wedge (\overline{f} \vee h)\big)$.

(Proof): See Appendix A.2.

Theorem 3.10. *The result of* $\texttt{ITE}(A, B, C)$ *is a reduced ordered ZDD,if* A, B *and* C *are reduced ordered ZDDs.*

Proof. For case 1, 2, 3 and 4 it is true, since A, B and C are reduced ordered ZDDs. For case 5, according to Section 3.5 the result of $\texttt{Not}(A)$ is a reduced ordered ZDD, if A is a reduced ordered ZDD, which is true based on the definition. In the other cases, the result of $< x, F_{x=1}, F_{x=0} >$ is a reduced ordered ZDD, if both $F_{x=0}$ and $F_{x=1}$are ordered ZDDs, and if

x is less than all variables in $F_{x=0}$ and $F_{x=1}$. As A, B and C are ordered ZDDs, $A_{x=v}, B_{x=v}$ and $C_{x=v}$ are also reduced ordered ZDDs. x is the top variable of Σ, so it is also the top variable of A, B and C. Therefore, all variables in $A_{x=v}, B_{x=v}$ and $C_{x=v}$, which are reduced ordered ZDDs, are more than x. Based on this, we can conclude that $F_{x=0}$ and $F_{x=1}$ in cases 6 to 10 are ordered and all of their variables are greater than x. Then the result of `CN` is a reduced ordered ZDD, so the `ITE` result is also a reduced ordered ZDD. □

Implementation. The `ITE` implementation is as follows

Algorithm 2 `ITE` implementation

```
function ITE($A$,$B$,$C$)
                                                            ▷ Terminal cases
    if $A$ = zddZero($A.dom$) then return $C$
    if $A$ = zddOne($A.dom$) then return $B$
    if $B = C$ then return $B$
    if ($B$ = zddOne($A.dom$) and $C$ = zddZero($A.dom$)) then return $A$
    if ($B$ = zddZero($A.dom$) and $C$ = zddOne($A.dom$)) then return N($A$)
    if $A = C$ then return $C$ = zddZero($A.dom$)
    if $A = B$ then return $B$ = zddOne($A.dom$)
                                                            ▷ Cache Checking
    if IsInCache(ITE, $A, B, C$) then return $result$
                                               ▷ Remove top variable from inputs
    $x \leftarrow$ TopVar($A.dom$)
    $A_l \leftarrow$ Lowedge($A, x$) , $B_l \leftarrow$ Lowedge($B, x$) , $C_l \leftarrow$ Lowedge($C, x$)
    $A_h \leftarrow$ Highedge($A, x$), $B_h \leftarrow$ Highedge($B, x$), $C_h \leftarrow$ Highedge($C, x$)
                                                            ▷ Top node calculation
    $A_t \leftarrow$ TopNode($A$) , $B_t \leftarrow$ TopNode($B$) , $C_t \leftarrow$ TopNode($C$)

                                                            ▷ Recursive calculation
    if ($x = B_t$ and $x \neq C_t$) or ($x \neq A_t$ and $x \neq B_t$ and $x \neq C_t$) then
        $R_h \leftarrow$ zddZero($A.dom - x$)
    if ($x \neq A_t$ and $x = C_t$) then $R_h \leftarrow C_h$
    if ($x = A_t$ and $x = B_t$ and $x \neq C_t$) then $R_h \leftarrow$ ITE($A_h, B_h$, zddZero($A.dom - x$))
    if ($x = A_t$ and $x \neq B_t$ and $x = C_t$) then $R_h \leftarrow$ ITE($A_h$, zddZero($A.dom - x$), $C_h$)
    if ($x = A_t$ and $x = B_t$ and $x = C_t$) then $R_h \leftarrow$ ITE($A_h, B_h, C_h$)
    $R_l \leftarrow$ ITE($A_l, B_l, C_l$)
                                                            ▷ $result$ calculation
    $result \leftarrow$ CN($x, R_h, R_l$)
                                                            ▷ Add $result$ to cache
    PutInCache(ITE, $A, B, C, result$)
    return $result$
```

In the implementation of `ITE` we use caching to reduce the number of function calls, to make it more efficient. Also we use the following rules to simplify the calculation for some special cases. These rules help to find the result without calculation or improve using cached data.

$$\texttt{ITE}(A, B, A) \rightarrow \texttt{ITE}(A, B, 0_\Sigma)$$
$$\texttt{ITE}(A, A, C) \rightarrow \texttt{ITE}(A, 1_\Sigma, C)$$

In BDD operations we usually check if the variables are equal to 0 or 1, which sometimes helps to stop unnecessary further calculations. As we have seen in this section, we also used these values as an input of `ITE` operation to calculate different Boolean operations. In BDDs Boolean function $f = 0$ and $f = 1$ represented by 0 and 1 terminal, respectively. But based on ZDD definition Boolean formula $f = 1$, represents as a complete graph including all variables in ZDD domains. Creating this diagram needs some calculation, which we called the operation `zddOne`.

We implemented the `Or` operation that calculated using `ITE` with a B parameter equal to 1_Σ. We tried to avoid extra computation by re-implementing these operations. The `Or` definition is the same as `ITE`, the only difference is that we knew $B = 1_\Sigma$.

3.5 `Not` operation

Another important ZDD operation is negation. This operation calculate the complement of a ZDD in a specific domain. It has the type and specification

$$\texttt{Not (N)} : \mathbb{Z}_\Sigma \rightarrow \mathbb{Z}_\Sigma$$
$$\texttt{N}(\mathcal{Z}_\Sigma(f)) = \mathcal{Z}_\Sigma\left(\overline{f}\right)$$

Definition 3.11. Let v_Σ be shorthand for $\mathcal{Z}_\Sigma(v), v \in \{0, 1\}$, x be the top variable in Σ, $\Sigma' = \Sigma - x$, and let $A = \mathcal{Z}_\Sigma(f)$. Then N is defined as follows:

$$\texttt{N}(A) = \begin{cases} 0_\Sigma & A = 1_\Sigma \\ 1_\Sigma & A = 0_\Sigma \\ < x, 1_{\Sigma'}, \texttt{N}(A_{x=0}, \Sigma') > & x \neq A.t \\ < x, \texttt{N}(A_{x=1}, \Sigma'), \texttt{N}(A_{x=0}, \Sigma') > & x = A.t \end{cases}$$

Theorem 3.12. *The result of* $\texttt{N}(\mathcal{Z}_\Sigma(f))$ *is equal to* $\mathcal{Z}_\Sigma\left(\overline{f}\right)$.

(Proof): See Appendix A.3.

Theorem 3.13. *The result of* $\texttt{N}(A)$ *is a reduced ordered ZDD, if* A *is a reduced ordered ZDD.*

Proof. For case 1 and case 2, it is true, since 0_Σ and 1_Σ are by definition reduced ordered ZDDs. In cases 3 and 4 the `CreateNode` operation is used, so the result is a reduced ordered ZDD, if in case 3 x is less than all variables in $1_{\Sigma'}$ and $\texttt{N}(A_{x=0}, \Sigma')$. And in case 4 x is less than all variables in $\texttt{N}(A_{x=1}, \Sigma')$ and $\texttt{N}(A_{x=0}, \Sigma')$. In addition, $\texttt{N}(A_{x=0}, \Sigma')$ and $\texttt{N}(A_{x=1}, \Sigma')$ should also be reduced ordered ZDD. As A is an ordered ZDD, $A_{x=v}$ is also ordered ZDD. x is the top variable of A, so all variables in $A_{x=v}$,which is represented by ordered ZDD, are more than x. So x is less than all variables in $\texttt{N}(A_{x=v}, \Sigma')$. x is also less than variables in $1_{\Sigma'}$, since it only can contain variables in Σ', that are greater than x as mentioned before. Therefore the result of $\texttt{N}(A)$ is a reduced ordered ZDD. □

Implementation. The algorithm 3 shows `Not` implementation.

Algorithm 3 `Not` implementation

```
function N($A$)
                                                        ▷ Terminal cases
    if $A = \texttt{zddOne}(A.dom)$ then return $\texttt{zddZero}(A.dom)$
    if $A = \texttt{zddZero}(A.dom)$ then return $\texttt{zddOne}(A.dom)$
                                                        ▷ Cache Checking
    if $\texttt{IsInCache}(\texttt{N}, A)$ then return $result$
                                          ▷ Remove top variable from inputs
    $x \leftarrow \texttt{TopVar}(A.dom)$
    $A_l \leftarrow \texttt{Lowedge}(A, x)$ ,$A_h \leftarrow \texttt{Highedge}(A, x)$
    $A_t \leftarrow \texttt{TopNode}(A)$
                                                   ▷ Recursive calculation
    if $x < A_t$ then $R_h \leftarrow 1_\Sigma$
    if $x = A_t$ then $R_h \leftarrow \texttt{N}(A_h)$
    $R_l \leftarrow \texttt{N}(A_l)$
                                                      ▷ $result$ calculation
    $result \leftarrow \texttt{CN}(x, R_h, R_l)$
                                                    ▷ Add $result$ to cache
    $\texttt{PutInCache}(\texttt{N}, A, result)$
    return $result$
```

3.6 `Exist` (∃) operation

One of the methods used in the reachability algorithm is quantification. Both ∃ and ∀ can be calculated by `Exist (EX)` operation, which has the specification

$$\texttt{EX} : \mathbb{Z}_\Sigma, x : \mathbb{V} \rightarrow \mathbb{Z}_{\Sigma - x}$$
$$\texttt{EX}(\mathcal{Z}_\Sigma(f), X) = \mathcal{Z}_{\Sigma - X}(\exists X.f)$$

By using `EX` and `Not` operations, ∀ can be calculated using the rule

$$\forall X.A = \exists X.\texttt{N}(A)$$

Definition 3.14. Let $A = \mathcal{Z}_\Sigma(f)$ be a well-formed ZDD, v_Σ be shorthand for $\mathcal{Z}_\Sigma(v)$ and let $\texttt{EX}_{x=v}$ be shorthand for $\texttt{EX}(A_{x=v}, \Sigma', X)$, where $v \in \{0,1\}$ and $\Sigma' = \Sigma - x$. Also let x be the top variable in domain Σ. Then EX will be defined as follows:

$$\texttt{EX}(A,\Sigma,X) = \begin{cases} 0_{\Sigma-X} & A = 0_\Sigma \\ 1_{\Sigma-X} & A = 1_\Sigma \\ \texttt{ITE}(\texttt{EX}_{x=1}, 1_{\Sigma'-X}, \texttt{EX}_{x=0}, \Sigma' - X) & A.t = x \wedge x \in X \\ < x, \texttt{EX}_{x=1}, \texttt{EX}_{x=0} > & A.t = x \wedge x \notin X \\ < x, 0, \texttt{EX}_{x=0} > & A.t \neq x \wedge x \notin X \\ \texttt{EX}_{x=0} & A.t \neq x \wedge x \in X \end{cases}$$

Theorem 3.15. *The result of* $\texttt{EX}(\mathcal{Z}_\Sigma(f), X)$ *is equal to* $\mathcal{Z}_{\Sigma-X}(\exists X.f)$.

(Proof): See Appendix A.4.

Theorem 3.16. *The result of* $\texttt{EX}(A, X)$ *is a reduced ordered ZDD, if* A *is a reduced ordered ZDD.*

Proof. For case 1 and case 2, it is true, since $0_{\Sigma-X}$ and $1_{\Sigma-X}$ are by definition reduced ordered ZDDs. For case 3, the result of ITE is reduced and ordered, if $\texttt{EX}_{x=0}$ and $\texttt{EX}_{x=1}$ are reduced ordered ZDDs. $\texttt{EX}_{x=v}$,$v \in \{0,1\}$ is reduced and ordered, since $A_{x=v}$ is a reduced ordered ZDDs. So it is true for case 3 as well. In addition to this condition, if x is less than all variables in $\texttt{EX}_{x=0}$ and $\texttt{EX}_{x=1}$, the theorem is also true for case 4. As $A_{x=v}$ is a reduced ordered ZDD and x is the top variable in A, x is less than all variables in $A_{x=v}$. It is also true for the last case, since $\texttt{EX}_{x=0}$ is reduced and ordered. □

Implementation. The algorithm 4 shows EX implementation. In this implementation, following rule is considered to use cached date more efficient. The rule stops the calculation when there is not any variable to be quantified.

$$\texttt{EX}(A, \emptyset) \to A$$

3.7 Rename operation

Renaming some variables with another is another operation that is used in reachability algorithm. It is usually assumed that variables are renamed with new variables that are not in current domain. The Rename (R) operation for ZDD has the specification

$$\texttt{R} : \mathbb{Z}_\Sigma, (x : \mathbb{V}) \subseteq \Sigma, (x' : \mathbb{V}') \cap \Sigma = \emptyset \to \mathbb{Z}_{(\Sigma-x)\cup x'}$$
$$\texttt{R}(\mathcal{Z}_\Sigma(f), x, x') = \mathcal{Z}_{(\Sigma-x)\cup x'}(f)$$

Algorithm 4 `Exist` implementation

```
function EX($A, X$)
                                                              ▷ Terminal cases
    if $A =$ zddZero($A.dom$) then return zddZero($A.dom - X$)
    if $A =$ zddOne($A.dom$) then return zddOne($A.dom - X$)
    if $X = 0$ then return $A$
                                                              ▷ Cache Checking
    if IsInCache(EX, $A, X$) then return $result$
                                              ▷ Remove top variable from inputs
    $x \leftarrow$ TopVar($A.dom$)
    $A_l \leftarrow$ Lowedge($A, x$) ,$A_h \leftarrow$ Highedge($A, x$)
    $A_t \leftarrow$ TopNode($A$)
                                                         ▷ Recursive calculation
    $R_l \leftarrow$ EX($A_l, X - x$)
    if $x = A_t$ then $R_h \leftarrow$ EX($A_h, X - x$)
                                                          ▷ $result$ calculation
    if ($x = A_t$ and $x \in X$) then $result \leftarrow$ ITE($R_h$, zddOne($A.dom - X$), $R_l$)
    if ($x = A_t$ and $x \notin X$) then $result \leftarrow$ CN($x, R_h, R_l$)
    if $x \neq A_t$ then
        $R_l.dom = A.dom$
        $result \leftarrow R_l$
                                                      ▷ Add $result$ to cache
    PutInCache(EX, $A, X, result$)
    return $result$
```

Each variable can be replaced with any variable, and the `CreateNode` does not support ordering. Which means the new variable may be placed with in a wrong ordering. So we assume the new variables have the same ordering as before. For example, if all the variables in the domain are odd, each of them can rename with its next even number, that results an ordered ZDD. The case that each variable can be replaced with any other variable is also possible. But then we should use `ITE` instead of `CreateNode` to order variables, which needs inputs with the same domain. So we should extend the domain of each new variable, which makes the calculation more complex and inefficient.

Definition 3.17. Let $A = \mathcal{Z}_{\Sigma}(f)$ be a well-formed ZDD, where x is the top level of Σ, and let y be the top variable of A. We also assume that substituting x with x' does not change the ordering. Then `R` will be defined as follows:

$$\mathtt{R}(A, x, x') = \begin{cases} \mathcal{Z}_{(\Sigma - x)\cup x'}(f[x/x']) & A = 0_{\Sigma} \vee A = 1_{\Sigma} \\ < y, \mathtt{R}(A_{y=1}, x, x'), \mathtt{R}(A_{y=0}, x, x') > & y < x \\ < x', A_{x=1}, A_{x=0} > & y = x \\ < x', 0, A_{x=0} > & y > x \end{cases}$$

Theorem 3.18. *The result of* $\mathtt{R}(\mathcal{Z}_{\Sigma}(f), x, x')$ *is equal to* $\mathcal{Z}_{(\Sigma - x)\cup x'}(f[x/x'])$.

(Proof): See Appendix A.5.

Theorem 3.19. *The result of* $\mathtt{R}(A, x, x')$ *is a reduced ordered ZDD, if* A *is a reduced ordered ZDD.*

Proof. For case 1 it is true, since the result is 0_Σ or 1_Σ which are by definition reduced ordered ZDDs. The result in case 2 is an ordered ZDD, if y be less than all variables in $\mathtt{R}(A_{y=v}, x, x')$, and if $\mathtt{R}(A_{y=v}, x, x')$ is an ordered ZDD. Since A is ordered, $A_{x=v}$ is also anordered ZDD.

For case 3 it is true, if $A_{x=v}$ is ordered and all variables in $A_{x=v}$ are greater than x'. As mentioned before $A_{x=v}$ is ordered. We assumed x' can is used with the same order as x, then if x is less than all variables in $A_{x=v}$, x' should have the same property. As $A_{x=v}$ is ordered and $x = y$ is the top variable of A, then all variables in $A_{x=v}$ are greater than x. Therefore there are greater than x', too. In case 4, x' is suppressed because 1-edge is pointing to zero. So x' does not effect on ordering. The theorem is also true for this case, if the result of $\mathtt{EX}(A, \Sigma, x)$ is reduced ordered ZDD, which is true since A is a reduced ordered ZDD.

□

Implementation. The algorithm 5 shows `Rename` implementation. In the implementation, the case that input ZDD is equal to terminal node 1 is also considered, since the representation does not change by renaming a variable and the domain should only become updated.

3.8 `RelProd` operation

Reachability algorithm which is explained in chapter 2.1.1, used three steps to find the next reachable states using current reached states S_Σ and transition relations $T_{\Sigma,\Sigma'}$. First step is finding possible transitions from current states. This can be calculated using $S_\Sigma \wedge T_{\Sigma,\Sigma'}$, as it limits transitions. The reachable states $S'_{\Sigma'}$ is the result of extracting the Σ variables from the result, by $\exists\Sigma$. This represents the next states in domain Σ', which should be renamed with variables in Σ.

Usually the first two steps are calculated in one, using the relational product of current states S_Σ and transition relation $T_{\Sigma,\Sigma'}$. For Boolean functions f and g , and Σ as a set of variables the relational product is defined as, $\exists\Sigma(f \wedge g)$. The `RelProd`(`RP`) operation for ZDD has the type and specification

$$\mathtt{RP} : \mathbb{Z}_\Sigma, \mathbb{Z}_\Sigma, (x : \mathbb{V}) \rightarrow \mathbb{Z}_{\Sigma-X}$$
$$\mathtt{RP}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_\Sigma(h), X) = \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))$$

Algorithm 5 Rename implementation

```
function RENAME($A, x, x'$)
                                                        ▷ Terminal cases
    if $A = \texttt{zddZero}(A.dom)$ then return $\texttt{zddZero}((A.dom - x) \cup x')$
    if $A = \texttt{zddOne}(A.dom)$ then return $\texttt{zddOne}((A.dom - x) \cup x')$
    if $A = 1$ then                                     ▷ Terminal node 1
        $A.dom = (A.dom - x) \cup x'$
        return $A$
                                                        ▷ Cache Checking
    if IsInCache(Rename, $A, x, x'$) then return $result$
                                                        ▷ Remove top node from input
    $A_t \leftarrow$ TopNode($A$)
    $A_l \leftarrow$ Lowedge($A, A_t$) ,$A_h \leftarrow$ Highedge($A, A_t$)
                                                        ▷ Recursive calculation
    if $x > A_t$ then
        $R_h \leftarrow$ Rename($A_h, x, x'$)
        $R_l \leftarrow$ Rename($A_l, x, x'$)
                                                        ▷ $result$ calculation
    if $x > A_t$ then $result \leftarrow$ CN($A.t, R_h, R_l$)
    if $x = A_t$ then $result \leftarrow$ CN($x', A_h, A_l$)
    if $x < A_t$ then
        $result \leftarrow EX(A, x)$
        $result.dom \leftarrow result.dom \cup x'$
                                                        ▷ Add $result$ to cache
    PutInCache(Rename, $A, x, x', result$)
    return $result$
```

Definition 3.20. Let $A = \mathcal{Z}_{\Sigma}(f)$ and $B = \mathcal{Z}_{\Sigma}(h)$ be well-formed ZDDs, $\texttt{RP}_{x=v}$ be the shorthand for $\texttt{RP}(A_{x=v}, B_{x=v}, \Sigma', X)$, and v_{Σ} be the shorthand for $\mathcal{Z}_{\Sigma}(v), v \in \{0,1\}$. Let x be the top variable in domain Σ. Then the definition of RP is as follows:

$$\texttt{RP}(A, B, \Sigma, X) = \begin{cases} 1_{\Sigma - X} & A = 1_{\Sigma} \wedge B = 1_{\Sigma} \\ 0_{\Sigma - X} & A = 0 \vee B = 0 \\ \texttt{ITE}(\texttt{RP}_{x=0}, 1_{\Sigma - X}, \texttt{RP}_{x=1}) & x \in X \\ < x, \texttt{RP}_{x=1}, \texttt{RP}_{x=0} > & x \notin X \wedge x = A.t \wedge x = B.t \\ < x, 0, \texttt{RP}_{x=0} > & \text{Otherwise} \end{cases}$$

Note. In RelProd definition, it is assumed that A and B have the same domain Σ.

Theorem 3.21. *The result of* $\texttt{RP}(\mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma}(h), X)$ *is equal to* $\mathcal{Z}_{\Sigma - X}(\exists X.(f \wedge h))$.

(Proof): See Appendix A.6.

Theorem 3.22. *The result of* $\texttt{RP}(A, B, X)$ *is a reduced ordered ZDD, if A and B are reduced ordered ZDDs.*

Proof. For case 1 and case 2 it is true, since $0_{\Sigma-X}$ and $1_{\Sigma-X}$ are by definition reduced ordered ZDDs. For case 3, it is also true, if $A_{x=v}$ and $B_{x=v}$ are reduced ordered ZDDs, which is true since A and B are reduced and ordered. Therefore the result of $\texttt{RP}_{x=v}$ is also represented by a reduced ordered ZDD, so the result of `ITE` is also ordered and reduced. In the other two cases, the `CreateNode` result is reduced and ordered, if x is less than the variables in $\texttt{RP}_{x=v}$, and if $\texttt{RP}_{x=v}$ is a reduced ordered ZDD, which has been discussed already. $A_{x=v}$ and $B_{x=v}$ are reduced ordered ZDDs and x is the top variable of A and B, so x is less than all variables in $A_{x=v}$ and $B_{x=v}$. As a result, x is also less than all variables of $\texttt{RP}_{x=v}$. Therefore, the result of `RP` is a reduced ordered ZDD for these cases as well. □

Implementation. The algorithm 6 shows `RelProd` implementation.

Algorithm 6 `RelProd` implementation

```
function RELPROD($A, B, X$)
                                                              ▷ Terminal cases
    if ($A$ = zddZero($A.dom$) or $B$ = zddZero($A.dom$)) then
        return zddZero($A.dom - X$)
    if ($A$ = zddOne($A.dom$) and $B$ = zddOne($A.dom$)) then
        return zddOne($A.dom - X$)
                                                              ▷ Cache Checking
    if IsInCache(RelProd, $A, B, X$) then return $result$
                                                    ▷ Remove top node from input
    $x \leftarrow$ TopVar($A.dom$)
    $A_l \leftarrow$ Lowedge($A, x$) , $B_l \leftarrow$ Lowedge($B, x$)
    $A_h \leftarrow$ Highedge($A, x$), $B_h \leftarrow$ Highedge($B, x$)

    if $x \in X$ then                          ▷ Recursive calculation for $x \in X$
        $R_l \leftarrow$ RelProd($A_l, B_l, X$)
        if $R_l$ = zddOne($A.dom - X$) then return zddOne($A.dom - X$)
        else
            $R_h \leftarrow$ RelProd($A_h, B_h, X$)
            $result \leftarrow$ Or($R_l, R_h$)
    else                                       ▷ Recursive calculation for $x \notin X$
        $R_l \leftarrow$ RelProd($A_l, B_l, X$)
        if ($x = A_t$ and $x = B_t$) then $R_h \leftarrow$ RelProd($A_h, B_h, X$)
        else$R_h = 0$
        $result \leftarrow$ CN($x, R_h, R_l$)
                                                      ▷ Add $result$ to cache
    PutInCache(RelProd, $A, B, X, result$)
    return $result$
```

3.9 `RelProdS` operation

In [29], `RelProdS` operation is introduced, which calculates both relational product (`RelProd`) and substitution(`Rename`).

The `RelProdS(RPS)` operation for ZDD has the type and specification

$$\texttt{RPS} : \mathbb{Z}_\Sigma, \mathbb{Z}_{\Sigma'}, (x : \mathbb{V}), (s : \langle \mathbb{V}', \mathbb{V}'' \rangle) \rightarrow \mathbb{Z}_{(\Sigma \cup \Sigma' - X)[S]}$$
$$\texttt{RPS}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_{\Sigma'}(h), X, S) = \mathcal{Z}_{(\Sigma \cup \Sigma' - X)[S]}(\exists X.(f \wedge h)[S])$$

Definition 3.23. Let $A = \mathcal{Z}_\Sigma(f)$, $B = \mathcal{Z}_{\Sigma'}(h)$ be well-formed ZDDs, and $\texttt{RPS}_{x=v}$ be the shorthand for $\texttt{RPS}(A_{x=v}, B_{x=v}, X, S)$, v_Σ be the shorthand for $\mathcal{Z}_\Sigma(v), v \in \{0, 1\}$, and $x = max(\Sigma.t, \Sigma'.t)$. It is also assumed that renaming each x with $S(x)$ will not change the ordering. Then the definition of `RPS` is as follows:

$$\texttt{RPS}(A, B, X, S) = \begin{cases} 1_{(\Sigma \cup \Sigma' - X)[S]} & A = 1_\Sigma \wedge B = 1_{\Sigma'} \\ 0_{(\Sigma \cup \Sigma' - X)[S]} & A = 0_\Sigma \vee B = 0_{\Sigma'} \\ \texttt{ITE}(\texttt{RPS}_{x=0}, 1_{(\Sigma \cup \Sigma' - X - x)[S]}, \texttt{RPS}_{x=1}) & x \in X \\ < S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0} > & \text{Otherwise} \end{cases}$$

Note. In `RelProdS` definition, A and B can have the different domains Σ and Σ', respectively.

Theorem 3.24. *The result of* $\texttt{RPS}(\mathcal{Z}_\Sigma(f), \mathcal{Z}_{\Sigma'}(h), X, S)$ *is equal to* $\mathcal{Z}_{(\Sigma \cup \Sigma' - X)[S]}(\exists X.(f \wedge h)[S])$.

(Proof): See Appendix A.7.

Theorem 3.25. *The result of* $\texttt{RPS}(A, B, X, S)$ *is a reduced ordered ZDD, if A and B be reduced ordered ZDDs.*

Proof. For case 1 and case 2 it is true, since $0_{(\Sigma \cup \Sigma' - X)[S]}$ and $1_{(\Sigma \cup \Sigma' - X)[S]}$ are by definition reduced ordered ZDDs. For case 3, it is also true, if $A_{x=v}$ and $B_{x=v}$ are reduced ordered ZDDs. This is true since A and B are by definition ordered and reduced. Therefore the result of $\texttt{RPS}_{x=v}$ is also represented by a reduced ordered ZDD, so the result of `ITE` is also ordered and reduced.

The result of case 4 is a reduced ordered ZDD, if $S(x)$ is less than all variables in $\texttt{RPS}_{x=v}$, and $\texttt{RPS}_{x=v}$ is represented by a reduced ordered ZDD. It is assumed that substituting x with $\S(x)$ does not change the ordering. Therefore, if x is less than all variables in $\texttt{RPS}_{x=v}$, then $S(x)$ is also less than them. $A_{x=v}$ and $B_{x=v}$ are reduced ordered ZDDs and x is the top variable of A and B, so x is less than all variables in $A_{x=v}$ and $B_{x=v}$. Therefore, x is also less than all variables of $\texttt{RP}_{x=v}$, which is the same for $S(x)$. Moreover, $\texttt{RPS}_{x=v}$ is represented by a reduced ordered ZDD, as discussed earlier, so the theorem is also true for the last case.

□

Implementation. The algorithm 7 shows `RelProdS` implementation. It is assumed that the expected result domain is pre-calculated, and is part of input variables, which is indicated as Σ. In this way we can check the cache without calculation.

Algorithm 7 `RelProdS` implementation

```
function RelProdS($A, B, X, S, \Sigma$)
                                                        ▷ Terminal cases
    if ($A$ = zddZero($A.dom$) or $B$ = zddZero($A.dom$)) then
        return zddZero($\Sigma$)
    if ($A$ = zddOne($A.dom$) and $B$ = zddOne($A.dom$)) then
        return zddOne($\Sigma$)
                                                        ▷ Cache Checking
    if IsInCache(RelProdS, $A, B, X, S, \Sigma$) then return $result$
                                                        ▷ Remove top node from input
    $x \leftarrow$ TopVar($A, B$)
    $A_l \leftarrow$ Lowedge($A, x$) , $B_l \leftarrow$ Lowedge($B, x$)
    $A_h \leftarrow$ Highedge($A, x$), $B_h \leftarrow$ Highedge($B, x$)

    if $x \in X$ then                                   ▷ Recursive calculation for $x \in X$
        $R_l \leftarrow$ RelProdS($A_l, B_l, X, S, \Sigma - x$)
        if $R_l$ = zddOne($\Sigma$) then return zddOne($\Sigma$)
        else
            $R_h \leftarrow$ RelProdS($A_h, B_h, X, S, \Sigma - x$)
            $result \leftarrow$ Or($R_l, R_h$)
    else                                                ▷ Recursive calculation for $x \notin X$
        $R_l \leftarrow$ RelProdS($A_l, B_l, X, S, (\Sigma - x) \cup S(x)$)
        $R_h \leftarrow$ RelProdS($A_h, B_h, X, S, (\Sigma - x) \cup S(x)$)
        $result \leftarrow$ CN($x, R_h, R_l$)
                                                        ▷ Add $result$ to cache
    PutInCache(RelProdS, $A, B, X, S, \Sigma, result$)
    return $result$
```

Chapter 4

Experiments

This chapter discusses our experiments. First the design of experiment for each set of models is explained. Then the result of these experiments are compared, for both cases of using BDD and ZDD implementation in Sylvan for finding the reachable state space of two sets of models: Sokoban puzzle and BEEM database [26]. In both cases we discuss how the memory and computation time varies using ZDDs and BDDs, in different steps of the algorithm. Some investigations are also done to improve the computation time of ZDDs.

4.1 Setups

The goal of this experiment is to investigate the efficiency of ZDDs in symbolic model checking. We used our implementation of ZDD operations and the existing BDD library in Sylvan. We measured computation time for each run. We also compared memory usage by measuring the size of used hash table and the number of nodes that are used for representing the final reachable state space, and transition relations for different models. We compared the size of these diagrams using BDDs and using ZDDs.

We did some measuring for each run, for example the size of reachable states in different iteration and the number of calling each function. Since ZDDs and BDDs represent the same formula with diagrams of different sizes, the calculation of execution time should exclude the time needed to compute the size of each diagram. So measuring execution time is done separately from measuring the memory usage. In addition, we used the gperftools CPU profiler [2] to measure and analyze the runtime behavior of the program.

Reachability algorithm We implemented the reachability algorithm, given in Algorithm 8 to find all reachable states in a model. We modify the algorithm in Section 2.1.1 using the `RelProdS` operation explained in Section 3.9.

Algorithm 8 Reachability algorithm

```
1: function REACHABILITY($I$,$T$,$\Sigma$,$\Sigma'$)
2:     $states, new \leftarrow I$
3:     while $new \neq \emptyset$ do
4:         $new \leftarrow$ RelProdS($T, \Sigma, \Sigma'$)
5:         $states \leftarrow states \vee new$
       return $states$
```

As we describe in Section 4.3, the BEEM models' transition relations are divided into couple of groups depending on the model. So the reachability algorithm is modified for this experiment to check all the transition groups in each iteration. There is a loop inside the main iteration, to calculate reachable states from each group of transitions. See Algorithm 9 for the modified version.

Algorithm 9 Modified version of reachability algorithm for BEEM models

```
1: function REACHABILITY($I$,$Tgroup$,$\Sigma$,$\Sigma'$)
2:     $states, new \leftarrow I$
3:     while $new \neq \emptyset$ do
4:         $old \leftarrow new, new \leftarrow \emptyset$,
5:         for $i = 1$ to $number\ of\ groups$ do
6:             $a \leftarrow$ Diff(RelProdS($old, Tgroup[i], \Sigma, \Sigma'$), $states$)
7:             $new \leftarrow new \vee a$
8:         $states \leftarrow states \vee new$
9:     return $states$
```

Impelementation We did our experiments based on the implemented ZDD operations using Sylvan library, explained in Chapter 2.5. ZDD operations are implemented as explained in Chapter 3. We extended the Sylvan library to support ZDDs as well, for symbolic model checking.

Models We executed the reachability algorithm on different models, to compare the efficiency of BDD and ZDD for symbolic model checking. We did our experiments on two sets of models, Sokoban puzzles and BEEM database [26]. Section 4.2 explains more about Sokoban puzzles and how we model them. The BEEM database is a database for explicit model checking. For each of them, we made a selection on models with different sizes of state space. Table 4.1 and 4.2 represent some information about the selected models. In this table, $Iterations$ is the number of needed iterations to find all reachable states in the reachability algorithm. $State$ is the number of reachable states in the model. $BDDnodes$ and $ZDDnodes$ are the number of

Model	Iterations	States	BDD nodes	ZDD nodes	% improvement
Screen.107	109	10,165	753	369	51.00%
Screen.1001	111	127,509	4080	2260	44.61%
Screen.387	172	1,235,214	4465	2706	39.40%
Screen.372	233	10,992,856	11578	6562	43.32%
Screen.792	174	117,434,655	32879	22546	31.43%
Screen.747	218	1,307,942,326	257344	196046	23.82%
Screen.38	267	12,197,960,188	130986	85183	34.97%

TABLE 4.1: Number of iterations for the reachability algorithm and number of nodes for BDD and ZDD representation of reachable states for Sokoban screens

Model	Group	Iterations	States	BDD nodes	ZDD nodes	% improvement
anderson.1	1292	6	352664	22221	6783	69.47%
anderson.6	180	18	18306917	75220	33328	55.69%
anderson.8	245	34	538699029	285064	125575	55.95%
scheduleworld.2	17	26	1570340	18779	3196	82.98%
scheduleworld.3	24	34	166649331	28500	5220	81.68%
at.5	50	33	31999440	156785	59699	61.92%
at.6	94	33	160589600	420526	174863	58.42%
at.7	56	33	819243816	986322	392439	60.21%
collision.4	167	37	41465643	38327	8001	79.12%
collision.5	180	37	4139765993	29537	8729	70.45%

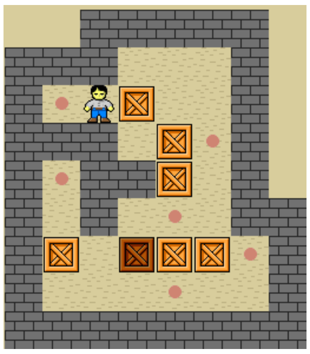
TABLE 4.2: Number of iterations and groups of transition relation for the reachability algorithm and number of nodes representing reachable states using BDD and ZDD for BEEM database models

required nodes to represent the final reachable states using BDD and ZDD, respectively. The last column, i.e., $improvement$ is the reduction percentage in number of nodes to represent reachable states using ZDD instead of BDD.

Architecture The experiments are performed on a cluster with 16 Intel Xeon X5550 @ 2.67GHz processors, each with 4 cores, and 72GB total memory .

4.2 Sokobon models

Sokoban is a puzzle, where a player should push some boxes to specific defined locations, i.e., goals, on a board. The board is partitioned in square cells, where each of them models either a floor or a wall. A floor cell may or may not contain a box, and player can move into a cell if it is empty or by pushing the box to an adjacent cell, if the cell is filled with the box. Moreover, boxes can not be pushed into walls or other boxes. The goal locations for boxes are also marked on some floor squares. The following is an example of Sokobon puzzle board.



In the first experiment, we used the reachability algorithm to find all possible positions for the player and boxes in the board, given their initial locations. Since there are four options for each squares, we need two Boolean variables x_1 and x_2 to represent the state of a cell, as described in Section 2.1. Here is how we encode the state of each cell:

Square condition	x_1x_2
empty	00
box	01
player	10
wall	11

We chose this encoding based on the definition of ZDD, that eliminates nodes with value 0. So we assigned $x_1 = 0$, $x_2 = 0$ to empty squares, since most of squares are empty. And because walls do not change and are not used in state specification, we assigned value 1 to the variables for representing walls ($x_1 = 1$, $x_2 = 1$). The other two possible evaluations also assigned to the players and boxes. So the number of ones in the representation of a state is equal to the total number of boxes and players in the board.

4.2.1 Results

Table 4.1 shows the size of BDDs and ZDDs representing the reachable set of states for different Sokoban models. As we expected ZDDs represent the same set of states, using less number of nodes in comparison to BDDs. We also keep track of the size of transition relation, during defining new transitions. Figure 4.1 represents the number of nodes used to represent transition relation in different steps of creating the final transition relation. We also measured how much memory is used by each method to complete reachability algorithm. As Table 4.3 shows the memory usage of ZDDs is about half of BDDs. Figure 4.2 represents the size of reachable states in each iteration of reachability algorithm, that also illustrates ZDDs efficiency in using memory in comparison with BDDs , for Sokoban puzzles.

As we have seen the size of transitions, the mid and final reachable state was smaller using ZDD representation instead of BDDs. So ideally we expected to have less computation time using

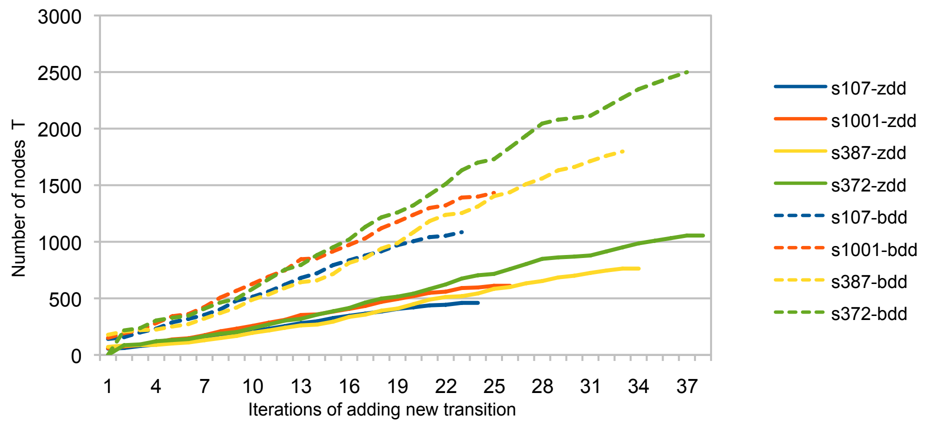
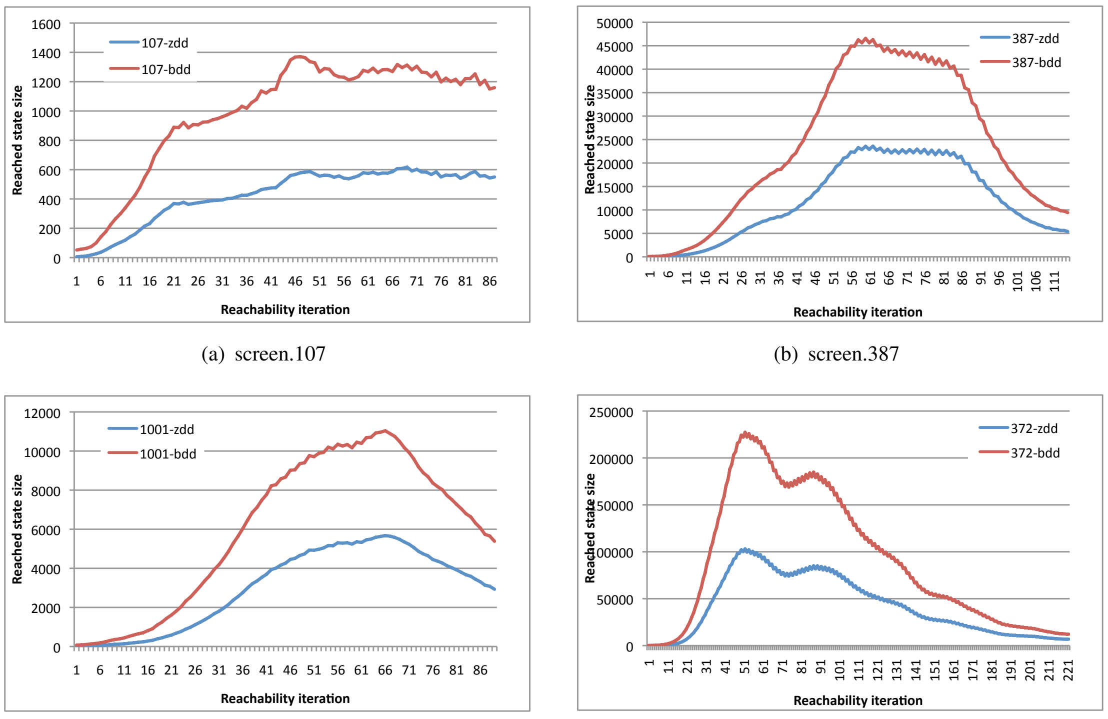


FIGURE 4.1: Transition relation size of Sokoban examples for ZDD and BDD



(a) screen.107

(b) screen.387

(c) screen.1001

(d) screen.372

FIGURE 4.2: Size of reached states in different iteration for BEEM models

Model	BDD	ZDD
screen.107	72,008	35,112
screen.1001	370,993	197,596
screen.387	2,200,884	1,149,400
screen.372	12,293,742	6,035,300
screen.792	132,756,758	76,052,580

TABLE 4.3: Used memory for Sokoban models in number of used buckets

Model	BDD	ZDD
screen.107	2,861	2,882
screen.1001	3,100	3,084
screen.387	4,846	4,614
screen.372	17,919	14,883
screen.792	184,411	156,642

TABLE 4.4: Computation time of Sokoban examples (ms)

Model	ZDD	BDD
screen.107	499,630	499,674
screen.1001	3,338,329	3,338,373
screen.387	20,261,938	20,263,440
screen.372	110,698,226	110,699,813
screen.792	1,380,276,935	1,379,945,068

TABLE 4.5: Number of times `RelProdS` operation called

ZDDs, but as Table 4.4 shows the computation time is almost the same using both methods. Although the memory consumption shows a great improvement, we need to investigate why computation time does not decrease when there are less number of nodes.

The following provides some possible explanation for why the computation time has not improved:

- The implemented operations for ZDDs are based on domain variables, which means the operations recursively call sub-graphs according to the variables in domain of a ZDD instead of variables showing up in the diagram. This may cause an effect on execution time, since although the ZDD is smaller, it has the same domain as BDD. We test this hypothesis by comparing the number of time each function is called using both methods. Table 4.5 illustrates how often `RelProdS` operation is called, as one of the main operation in reachability algorithm. Based on these results, the number of function calls are quite the same for ZDDs and BDDs, which may be the reason of having the same time.

  We also measure how much time the entire process has spent in each function. Table 4.6 shows the percentage of time spent in a certain operation during the experiment using ZDDs and BDDs. Since the number of used node is less in ZDD, `llmsset_lookup` operation that creates nodes in hash table, call 10% less, while it spend the same percentage of time for caching. Calculation of `RelProdS` and `ITE` for BDD, in total is almost the same as ZDD calculation of `RelProdS` and `Or`. The saved time for creating nodes, is spend for finding the level of a node and calculating the set without its top variable, which both are used for domain calculation. So there is a possibility that we can have speedup for ZDDs by optimizing domain calculation.

operation	BDD%	ZDD%	description
`llmsset_lookup`	23.4	13.4	creating nodes in hash table
`llci_get_tag`	14.3	14.3	retrieving from memoization cache
`relprods`	16.4	20.5	calculating `RelProdS`
`hash_mul`	13.1	11.2	hashing function
`memcpy`	13.1	14.7	copy or fill block of memory
`ite`	9	-	calculating `ITE`
`or`	-	5.8	calculating `Or`
`__nss_hosts_lookup`	2.5	4.9	
`llci_put_tag`	2	1.8	putting in memoization cache
`sylvan_var`	-	3.6	get the level of a node
`sylvan_set_next`	-	3.1	calculate set without top variable
sum	93.8	93.3	

TABLE 4.6: CPU profile of screen.387 using ZDD and BDD

We also investigated the effect of using operations in which it recursively call sub-graphs according to the variables showing up in the diagram, for the BEEM database. The related result is represented in Section 4.3.1.

- Since its an ongoing work, some of our ZDD operations may not yet be as efficient as BDD operations of Sylvan. Therefore, we may have more execution time for ZDDs.

- Also, if we would use garbage collection, it would be possible to run large examples, otherwise the program usually gets out of memory. Garbage collection calls when the memory is quite full, and using it will take some time to clear the hash table. So if we use less memory, we need to call garbage collection less and save some time as a result in comparison to the same algorithm using more memory. So we expect less computation time for ZDD, since the memory usage is half of BDD. We did not use garbage collection in this experiment, and that's why it was not doable for larger examples. So there is a hypothesis that doing the same experiment on larger example and including garbage collection will result in a better execution time for ZDDs.

  We used garbage collection to find reachable states of these models, to check whether it can make a different. Table 4.7 represents the new results, that confirms the hypothesis of using garbage collection improve the result of ZDDs for larger examples.

4.3 BEEM models

We used the BEEM database [26], [1] models where transition relations are divided in to n groups, where n depends on the model . Algorithm 9 is used instead of the original reachability algorithm for these models, and find the reachable states in each iteration, in n steps.

Model	BDD	ZDD
screen.107	2,375	2,389
screen.1001	2,574	2,579
screen.387	4,187	3,984
screen.372	14,855	12,982
screen.792	174,551	147,008
screen.747	3,130,416	2,374,375
screen.38	15,657,673	10,241,960

TABLE 4.7: Computation time of Sokoban examples using garbage collection (ms)

First these transition relation groups are calculated using `LTSmin` [6], a model checking toolset. The full command is:

```
dve2lts-sym -rgs   --vset=sylvan <model.dve> --save-transitions
=<file.bdd>
```

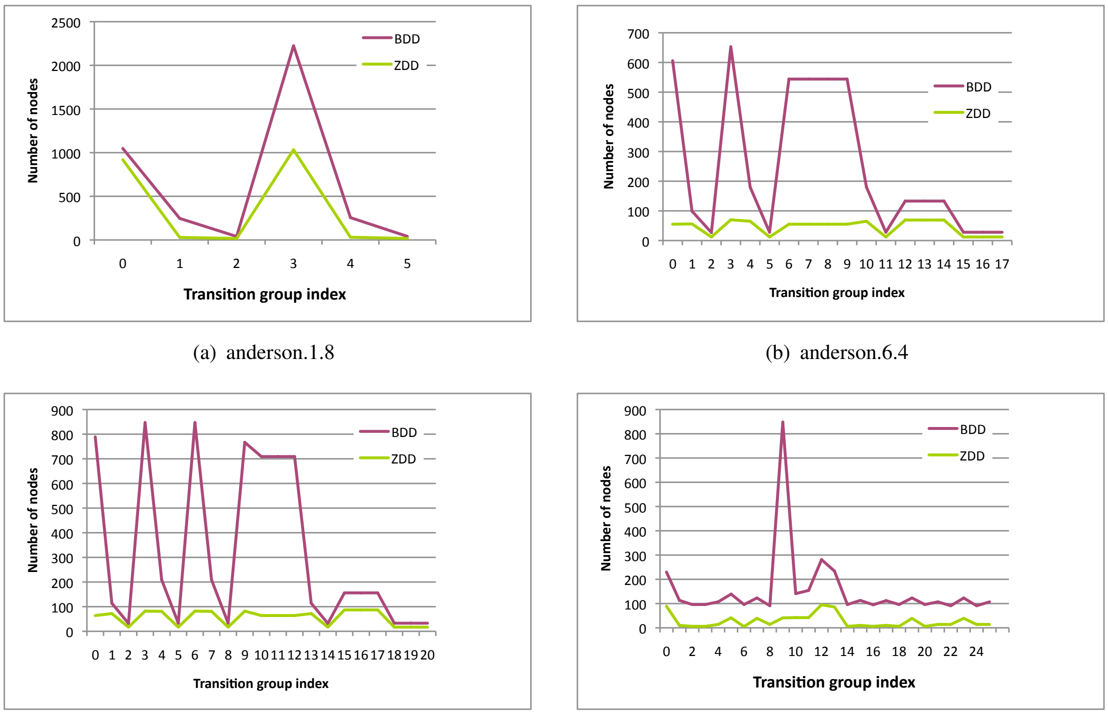
Then we generated the related BDDs for initial state and all transition relation groups, based on the generated `file.bdd`. After that, we converted BDDs to ZDDs using `B2Z` operation, as explained in Section 3.2.

In this experiment, we pre-computed the ZDD and BDD initial states and transition relation first, to focus on reachability algorithm. Otherwise, we would end up measuring the performance of generating the initial DDs and converting BDDs to ZDDs, which were not our interest. So the execution times used in the following is only related to reachability algorithm not generating diagrams.

4.3.1 Results

The size of BDDs and ZDDs representing the reachable set of states for different BEEM models are shown in Table 4.2. The same as Sokoban results, the number of nodes using ZDD is less than BDD to represent the set of reachable states. Figure 4.3 represent the size of each group of transition relation for couple of examples. As we can see, ZDD representations are smaller than BDDs for all groups. For some groups of transitions, BDDs are much larger than ZDDs. We investigated why we have these peaks in BDDs but not in ZDDs.

Each generated group of transitions for a model only includes variables that are used on the related Boolean formula. Which means don't care variables are not part of the domain. So the number of variables in the domain of each transition group can be different from the others. The existing peaks in BDD results, is related to transition groups with more variables. But most of them are assigned to zero that results in a major increase in the size of BDDs. However, it does not affect the size of ZDDs.



(a) anderson.1.8 (b) anderson.6.4

(c) anderson.8.4 (d) ScheduleWorld.2.8

FIGURE 4.3: Size of transition relation groups for BEEM models

We also measured the computation time of reachability algorithm for some of the BEEM database models. The columns BDD and ZDD-ITE in Table 4.9 depicts the results of this experiment, which for larger models ZDDs find all reachable states in less time in comparison to BDDs. This is, to some extent, different from the results that we had for Sokoban examples of previous section, for which we did not have a noticeable improvement in the computation time, in spite of the fact that the same data structure and operations are used for both cases.

As we mentioned earlier, there is a possibility that ZDDs have better performance when we use garbage collection, which clears the hash table when it is full. In this experiment we have used garbage collection. So the reduction of computation time could be linked to this fact. By measuring the number of times that the garbage collection (`gc`) is called for each model using ZDD and BDD, we can find out whether the improvement of execution time is related to `gc`. As it is shown in Figure 4.4 , there is a relation between the reduction in the number of times that the hash table becomes full (that results in calling `gc`) and the obtained speedup. Therefore the performance of ZDDs is better than BDDs, since they use less memory that results in calling `gc` less. Figure 4.5 represents the percentage of ZDD improvement in computation time versus to percentage of its improvement in number of calls to `gc` comparing with BDD.

Table 4.8 shows the percentage of time spent in a certain operation during the experiment using ZDDs and BDDs. The same as Sokoban example, ZDDs spent less time for creating nodes. However, in this case it use cached data more, and also it spend about two time more

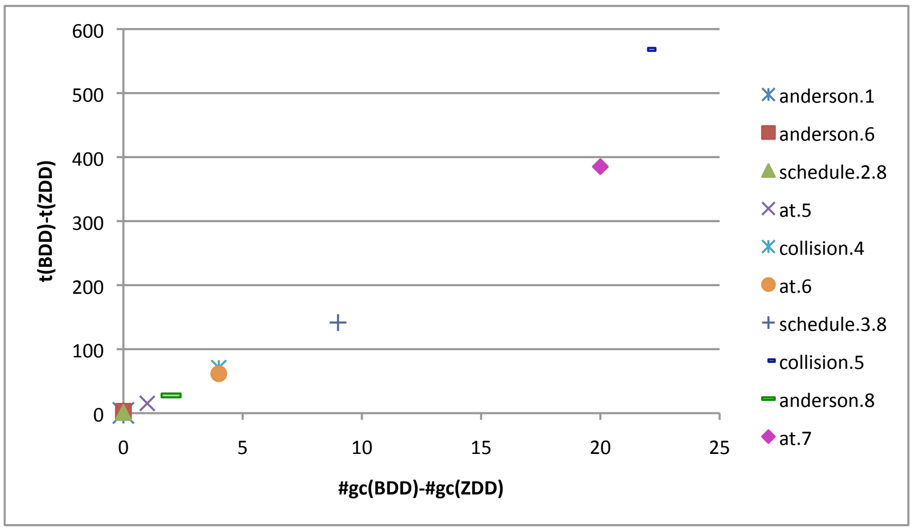


FIGURE 4.4: Correlation between the reduction of number of calls to `gc` and speedup, i.e., the decrease in computation time by using ZDD (s)

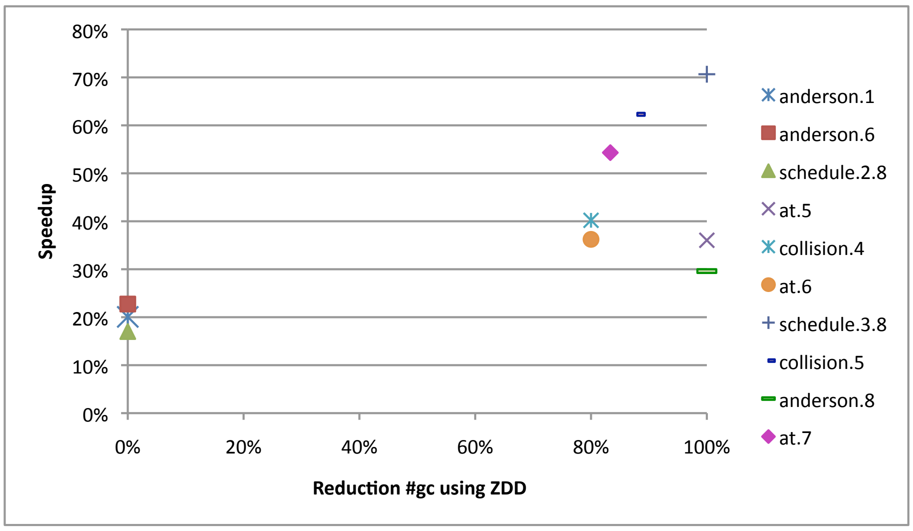


FIGURE 4.5: Speedup by percentage of reduction in number of times `gc` is called using ZDD

in `RelProdS` operation. Domain calculation in ZDD also takes about 7% of the computation time, using `sylvan_set_next` and `sylvan_var`.

Several implementations are investigated for ZDD operations, to reduce the ZDD computation time. Tables 4.9 and 4.10 show the results of different implementations, where *ZDD-ITE* shows the results for using `ITE` operation for `Or`, `And`, and `Diff`, which is the same as BDD. As mentioned in Section 3.4, there is also a separate implementation for `Or` operation, which prevents calling `zddOne` operation unnecessarily. The results of using this operation are represented as *ZDD-Or* in these figures. As explained in the Sokoban experiment, there is a possibility that the execution time can speed up by traversing through the diagram according to the appeared variables in the diagram itself instead of traversing through the domain variables. To assess this,

operation	BDD%	ZDD%	description
`llmsset_lookup`	36.2	11.9	creating nodes in hash table
`hash_mul`	18.2	19.2	hashing function
`ite`	13.4	7.7	calculating `ITE`
`or`	-	4.2	calculating `Or`
`llci_get_tag`	9.5	17.6	retrieving from memoization cache
`relprods`	6.9	15.9	calculating `RelProdS`
`__nss_hosts_lookup`	2.4	3.7	
`llci_put_tag`	2.2	3.7	putting in memoization cache
`llmsset_rehash_bucket`	1.1	-	mapping existing items to new buckets
`memset-cpy`	1.9	2.7	copy or fill block of memory
`sylvan_var`	-	5.3	get the level of a node
`sylvan_set_next`	-	1.9	calculate set without top variable
sum	92%	93.8%	

TABLE 4.8: CPU profile of schedule_world.3.8 using ZDD and BDD

Model	BDD	ZDD-ITE	ZDD-or	ZDD-skip
anderson.1	1.5	1.8	1.3	1.2
anderson.6	12.9	12.9	12.9	10.0
schedule.2.8	6.1	5.0	4.5	5.1
at.5	42.5	34.2	32.0	27.2
collision.4	176.9	134.2	125.0	105.8
at.6	170.1	133.7	126.1	108.5
schedule.3.8	200.4	83.7	85.6	58.8
collision.5	912.1	424.4	419.6	343.9
anderson.8	93.5	81.7	75.8	65.8
at.7	709.0	410.9	374.8	324.0

TABLE 4.9: Computation time of BEEM models using BDD and ZDD

we modified the `Or` operation as an example, to skip domain variables that do not show up in the diagram. *ZDD-skip* is related to the result of using this method for `Or` operation.

It is obvious from Table 4.10, that the number of function calls is reduced by skipping unused variables, since there are smaller numbers in column ZDD-skip in compare with both ZDD-ITE and ZDD-or. This also affects on the execution time and helps speeding up using ZDD. So we may have better performance using ZDD, by changing other operations as well. However, using a specific implementation for `Or` operation, does not have too much influence on both execution time and function calls. The tables also illustrates that there is a relation between the number of function calls and the execution time. By decreasing the number of function calls, the execution time decreases.

Model	BDD	ZDD-ITE	ZDD-or	ZDD-skip
anderson.1	2,944,976	5,331,802	3,054,581	2,475,173
anderson.6	30,002,641	35,015,784	33,405,964	22,727,626
schedule.2.8	16,952,579	19,119,358	18,359,734	9,417,068
at.5	100,578,435	114,291,168	101,179,670	69,713,839
collision.4	481,120,675	526,511,631	457,870,260	321,199,791
at.6	416,970,328	462,999,565	410,561,120	296,568,050
schedule.3.8	576,702,703	411,924,160	399,459,312	202,934,785
collision.5	2,487,369,371	1,836,328,817	1,681,405,138	1,122,717,622
anderson.8	202,249,412	226,623,320	225,108,906	153,652,632
at.7	1,644,914,192	1,376,672,146	1,187,195,396	852,809,898

TABLE 4.10: Number of function calls for `ITE` and `OR` operations using BDD and ZDD for BEEM models

Chapter 5

Related work

ZDDs are used in different areas. In [22], several applications of ZDD are mentioned. The first category is related to representing the unate cube sets [20], where only positive literals are allowed. Unate cube sets are used to solve problems like the *N-queen problem*, which can be solved N-times smaller by representing cube sets using ZDDs instead of BDDs. However, the *knight's tour problem* as another famous combinatorial chess problem, do not benefit from ZDDs. The solution of this problem contains 64 edges from the 156 edges, which is not a sparse graph. So we can expect that ZDDs do not have remarkable effect on it.

ZDDs are also used to represent the cube sets (two-level logic), where each cube is a combination of positive and negative literals for input variables. Boolean functions can be represented by cube sets. As explained in [22], cube sets can be used to generate multi-level logic for logic synthesis systems. The problem with this is that there are cases where the cube set representations grow exponentially with the number of inputs. For instance, parity functions and full-adders behave like this. Using the OBDD-based technique [11] makes an improvement in both computation time and memory usage. ZDDs are suitable for these kind of applications, where the multi-level logic can quickly be generated from cube sets, even for parity functions and full-adders.

In [31], a method for CTL model checking of Petri nets using ZDD is proposed. As the state space of Petri nets are usually sparse, ZDDs are useful for their manipulation. They also propose special operations to compute the reachable states for Petri nets. Their experimental results show that the size of ZDDs is two or three times smaller than the original BDDs, for symbolic manipulation of Petri nets.

Another usage of ZDDs is presented in [21], for representing polynomial formulas. The idea behind it is to break degrees and coefficients of each variable in to sum of 2's exponential numbers. For instance, the polynomial $5x^2 + 2x^{10}y$ can be written as $x^2 + 2^2x^2 + 2^1x^2x^8y^1$.

Then this combination can efficiently represented by ZDDs, using five variables 2^1, 2^2, x^2, x^8, and y^1.

Based on [25], ZDD is also used to different problems such as:

- To solve graph optimization problems like maximal clique [9].
- To represent cubes and essential primes in two-level SOP minimization [8].
- To solve unate covering problem arising in multi-layer planar routing [13] .
- To find dichotomy-based constraint encoding [12] [10].
- To represent and manipulate regular expressions under length constraint [15].
- In symbolic traversal of finite state machines [28].
- In pass-transistor logic synthesis [5].
- In finding all disjoint-support decomposition of completely specified logic functions [23].
- For unate decomposition of Boolean functions [16].

Chapter 6

Conclusions and Future Work

This chapter presents the conclusion of our work and some ideas as future challenges.

6.1 Conclusions

In Chapter 2, we introduced ZDD as a representation of Boolean functions, and explained the requirements of implementing reachability algorithm. Then we discussed about CUDD package as an available implementation of ZDD, and its strong and weak points to be used for model checking. CUDD is one of the known packages for BDDs, but it considers a fixed domain of variables for all diagrams, which is not suitable for ZDDs. There is also an extension for CUDD, called EXTRA, with more ZDD operations. However, still the same problem exist on EXTRA. We used some examples to explain this issue and some other reasons that we did not choose CUDD for our experiment.

In Chapter 3, we designed and implemented ZDD operations as an extension of Sylvan, such as `ITE`, `Exist`, and `RelProdS`. Moreover, we proved the correctness of these operations. Also a domain attribute was considered for each ZDD to avoid the problems on CUDD.

In chapter 4, we compared the performance of the reachability algorithm using ZDDs and BDDs, for a set of models in Sokoban puzzles and BEEM database. The results show an effective improvement in memory usage by using ZDDs. In fact the memory usage is about half time less than BDDs. ZDDs use less nodes to represent the same set of states, as well as transition relations. Moreover, the execution time also reduced for large cases. According to our experiments, since we need to use garbage collection for large models, ZDDs can save some time by calling garbage collection less than BDDs, which is a result of using less memory.

Additionally, we investigated how ZDDs can benefit from having less nodes to speedup. We found out the number of function calls is almost the same for both, in spite of the fact that the number of nodes decreased. This is happening since the functions are recursively called based on existing variables in the domain, which is the same for both ZDDs and BDDs. We modified one of the operations to skip variables that are in the domain but not in the diagram, to check whether it reduces the computation time. This resulted the speedup of ZDDs, which suggests that it may be beneficial to apply this modification on all operations.

6.2 Future Work

We also have some ideas for future investigation:

1. The computation time can be improved by skipping variables in the domain that are not used in the ZDD representation, while the caching is same as before. This will reduce the number of function calls.
2. Exploring the effect of using garbage collection for a larger set of Sokoban models, can prove or disprove improvement of ZDD execution time in comparison to BDD.
3. As we have seen in the experiments, about 7% of execution time is spent on operations that are mostly used for calculating the domain. Optimizing domain calculation may reduce the usage of these operations.
4. As mentioned earlier `zddOne` is not as trivial as it is for BDDs. Assume a ZDD with domain $X = \{x_1, ..., x_n\}$, If we store the domain variables using `zddOne(X)` instead of $x_1 \vee ... \vee x_n$, then we always have the result of `zddOne` for the related domain. This will avoid us from calculating it every time.
5. We can extend the current implementation of the operations so that they support inputs with different domains; specifically for operations like `ITE` and `RelProd`. It may reduce the number of calls to $Extend$ operation, which is used to match the domains. However, this extension does not affect on our experiment results, since `RelProdS` already supports different domains and we only use ITE for cases with the same domains.
6. Comparing CUDD and Sylvan-ZDD in both memory usage and computation time to investigate the effect of the domain attribute used in our implementation, would be interesting.
7. Parallelism of ZDD could improve its computation time.

Appendix A

Correctness Proofs

Based on the ZDD definition, we have the following rules, which are referred in the proofs:

R.1 $\mathcal{Z}_\Sigma(f) = 0 \longleftrightarrow f = 0$

R.2 $\mathcal{Z}_\Sigma(f) = 1_\Sigma \longleftrightarrow f = 1$

R.3 $\mathcal{Z}_\Sigma(f) = \mathcal{Z}_\Sigma(g) \longleftrightarrow f = g$

R.4 $(\Sigma.t = x \wedge A.t \neq x) \longleftrightarrow f_{x=1} = 0$

We also assumed following equivalence in the proofs:

$f \equiv (x \wedge f_{x=1}) \vee (\overline{x} \wedge f_{x=0})$	Shannon decomposition
$(f \wedge g)_{x=v} \equiv f_{x=v} \wedge g_{x=v}$	Distribution of restriction over $\wedge$
$(f \vee g)_{x=v} \equiv f_{x=v} \vee g_{x=v}$	Distribution of restriction over $\vee$
$\exists x.f \equiv f_{x=0} \vee f_{x=1}$	Definition of $\exists$
$\exists x.(f \vee g) \equiv \exists x.f \vee \exists x.g$	Distribution of $\exists$ over $\vee$
$(A - B) \cup C \equiv (A \cup C) - (B - C)$	Distribution of $\cup$ over $-$ for sets

A.1 `Extend` Operation

Definition A.1. Let $A = \mathcal{Z}_{\Sigma}(f)$, and let y be the top variable of A. Then `EXT` will be defined as follows:

$$\texttt{EXT}(A, x) = \begin{cases} A & x = y \\ < x, A, A > & x < y \\ < y, \texttt{EXT}(A_{y=1}, x), \texttt{EXT}(A_{y=0}, x) > & x > y \end{cases}$$

Theorem A.2. *The result of* $\texttt{EXT}(\mathcal{Z}_{\Sigma}(f), x)$ *is equal to* $\mathcal{Z}_{\Sigma \cup x}(f)$.

Proof. By induction on the size of A. We assume the induction hypothesis $\texttt{EXT}(\mathcal{Z}_{\Sigma - y}(f_{y=v}), x) \equiv \mathcal{Z}_{(\Sigma - y) \cup x}(f_{y=v})$, where $v \in \{0, 1\}$. There are five cases:

1. If $x = y$, then the function returns $A = \mathcal{Z}_{\Sigma}(f) = \mathcal{Z}_{\Sigma \cup x}(f)$, Since $y \in \Sigma$ and $x = y$ then $\Sigma \cup x = \Sigma$.

2. If $x < y$, then the function returns $< x, A, A >$

$$\begin{aligned} &\overset{1}{=} < x, \mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma}(f) > \\ &\overset{2}{=} \mathcal{Z}_{\Sigma \cup x}((x \wedge f) \vee (\overline{x} \wedge f)) \\ &\overset{3}{=} \mathcal{Z}_{\Sigma \cup x}(f) \end{aligned}$$

   Step 1: by definition of A. Step 2: by definition of `CreateNode`. Step 3: by logical equation.

3. If $x > y$, then the function returns $< y, \texttt{EXT}(A_{y=1}, x), \texttt{EXT}(A_{y=0}, x) >$

$$\begin{aligned} &\overset{1}{=} < y, \mathcal{Z}_{(\Sigma - y) \cup x}(f_{y=1})), \mathcal{Z}_{(\Sigma - y) \cup x}(f_{y=0})) > \\ &\overset{2}{=} \mathcal{Z}_{(\Sigma - y) \cup x \cup y}((y \wedge f_{y=1}) \vee (\overline{y} \wedge f_{y=0})) \\ &\overset{3}{=} \mathcal{Z}_{\Sigma \cup x}((y \wedge f_{y=1}) \vee (\overline{y} \wedge f_{y=0})) \\ &\overset{4}{=} \mathcal{Z}_{\Sigma \cup x}(f) \end{aligned}$$

   Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by logical equation. Step 4: by Shannon decomposition.

□

A.2 ITE Operation

Definition A.3. Assume $A = \mathcal{Z}_{\Sigma}(f), B = \mathcal{Z}_{\Sigma}(g), C = \mathcal{Z}_{\Sigma}(h)$, which are well-formed ZDDs . Let $A_{x=v}$ be shorthand for $\mathcal{Z}_{\Sigma'}(f_{x=v})$ and $\texttt{ITE}_{x=v}$ be shorthand for $\texttt{ITE}(A_{x=v}, B_{x=v}, C_{x=v})$ with $v \in \{0,1\}$. Also let x be top variable in Σ. Then ITE is defined as follows:

***Note*.** Note that in the following definition all cases except the terminal cases are the simplified version of

$$< x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0} >$$

$$\texttt{ITE}(A,B,C) = \begin{cases} C & A = 0_{\Sigma} \\ B & A = 1_{\Sigma} \\ B & B = C \\ A & B = 1_{\Sigma} \wedge C = 0_{\Sigma} \\ \texttt{Not}(A) & B = 0_{\Sigma} \wedge C = 1_{\Sigma} \\ < x, 0_{\Sigma'}, \texttt{ITE}_{x=0} > & (B.t \neq x \wedge C.t \neq x) \vee \\ & (A.t \neq x \wedge B.t = x \wedge C.t \neq x) \\ < x, C_{x=1}, \texttt{ITE}_{x=0} > & A.t \neq x \wedge C.t = x \\ < x, \texttt{ITE}(A_{x=1}, B_{x=1}, 0'_{\Sigma}), \texttt{ITE}_{x=0} > & A.t = x \wedge B.t = x \wedge C.t \neq x \\ < x, \texttt{ITE}(A_{x=1}, 0'_{\Sigma}, C_{x=1}), \texttt{ITE}_{x=0} > & A.t = x \wedge B.t \neq x \wedge C.t = x \\ < x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0} > & A.t = x \wedge B.t = x \wedge C.t = x \end{cases}$$

Theorem A.4. *The result of* $\texttt{ITE}(\mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma}(g), \mathcal{Z}_{\Sigma}(h))$ *is equal to* $\mathcal{Z}_{\Sigma}\big((f \vee g) \wedge (\overline{f} \vee h)\big)$.

Proof. By induction on the size of Σ. We assume the induction hypothesis
$\texttt{ITE}(A_{x=v}, B_{x=v}, C_{x=v}) \equiv \mathcal{Z}_{\Sigma'}\big((f_{x=v} \wedge g_{x=v}) \vee (\overline{f_{x=v}} \wedge h_{x=v})\big) \equiv \mathcal{Z}_{\Sigma'}\big(((f \wedge g) \vee (\overline{f} \wedge h))_{x=v}\big)$
(by distribution over restriction). There are nine cases:

1. If $A = 0_{\Sigma}$, then the function returns

$$\begin{aligned} C &\overset{1}{=} \mathcal{Z}_{\Sigma}(h) \\ &\overset{2}{=} \mathcal{Z}_{\Sigma}((0 \wedge g) \vee (1 \wedge h)) \\ &\overset{3}{=} \mathcal{Z}_{\Sigma}\big((f \wedge g) \vee (\overline{f} \wedge h)\big) \end{aligned}$$

Step 1: by definition of C. Step 2: by logic equivalent. Step 3: as $f = 0$, using rule **R.1**.

2. If $A = 1_\Sigma$, then the function returns

$$\begin{aligned} B &\overset{1}{=} \mathcal{Z}_\Sigma(g) \\ &\overset{2}{=} \mathcal{Z}_\Sigma((1 \wedge g) \vee (0 \wedge h)) \\ &\overset{3}{=} \mathcal{Z}_\Sigma\big((f \wedge g) \vee (\overline{f} \wedge h)\big) \end{aligned}$$

Step 1: by definition of B. Step 2: by logic equivalent. Step 3: as $f = 1$, using rule **R.2**.

3. If $B = C$, then the function returns

$$\begin{aligned} B &\overset{1}{=} \mathcal{Z}_\Sigma(g) \\ &\overset{2}{=} \mathcal{Z}_\Sigma\big((f \wedge g) \vee (\overline{f} \wedge g)\big) \\ &\overset{3}{=} \mathcal{Z}_\Sigma\big((f \wedge g) \vee (\overline{f} \wedge h)\big) \end{aligned}$$

Step 1:by definition of B. Step 2: by logic equivalent. Step 3: by rule **R.3**.

4. If $B = 1_\Sigma, C = 0_\Sigma$, then the function returns

$$\begin{aligned} A &\overset{1}{=} \mathcal{Z}_\Sigma(f) \\ &\overset{2}{=} \mathcal{Z}_\Sigma\big((f \wedge 1) \vee (\overline{f} \wedge 0)\big) \\ &\overset{3}{=} \mathcal{Z}_\Sigma\big((f \wedge g) \vee (\overline{f} \wedge h)\big) \end{aligned}$$

Step 1: by definition of A. Step 2: by logic equivalent. Step 3: by definition of B and C, and rules **R.1**, and **R.2**.

5. If $B = 0_\Sigma, C = 1_\Sigma$, then the function returns

$$\begin{aligned} \overline{A} &\overset{1}{=} \mathcal{Z}_\Sigma(\overline{f}) \\ &\overset{2}{=} \mathcal{Z}_\Sigma\big((f \wedge 0) \vee (\overline{f} \wedge 1)\big) \\ &\overset{3}{=} \mathcal{Z}_\Sigma\big((f \wedge g) \vee (\overline{f} \wedge h)\big) \end{aligned}$$

Step 1: by definition of A. Step 2: by logic equivalent. Step 3: by definition of B and C, and rules **R.1**, and **R.2**.

6. If $B.t \neq x, C.t \neq x$ Or $A.t \neq x, B.t = x, C.t \neq x$, then the function returns

$< x, 0'_{\Sigma}, \mathtt{ITE}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} && < x, 0'_{\Sigma}, \mathcal{Z}_{\Sigma'}\left(\left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right) > \\
&\overset{2}{=} && \mathcal{Z}_{\Sigma' \cup x}\left((x \wedge 0) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{3}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge 0) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{4}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge (0 \vee (\overline{f_{x=1}} \wedge 0)) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{5}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge (0 \vee (\overline{f_{x=1}} \wedge h_{x=1})) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{6}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge \left((f_{x=1} \wedge g_{x=1}) \vee (\overline{f_{x=1}} \wedge h_{x=1})\right)) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{7}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=1}) \vee \left(\overline{x} \wedge \left((f \wedge g) \vee (\overline{f} \wedge h)\right)_{x=0}\right)\right) \\
&\overset{8}{=} && \mathcal{Z}_{\Sigma}\left((f \wedge g) \vee (\overline{f} \wedge h)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by assumption A, B, and C are well-formed ZDDs. Step 4: by logical equation. Step 5: as $C.t \neq x$, x has negative value in h (rule **R.4**). Step 6: as $A.t \neq x$ Or$B.t \neq x$, x has negative value in f or g. Step 7: distribution of restriction. Step 8: by Shannon decomposition.

7. If $A.t \neq x, C.t = x$, then the function returns

   $< x, C_{x=1}, \mathtt{ITE}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} && < x, \mathcal{Z}_{\Sigma'}\left(h_{x=1}\right), \mathcal{Z}_{\Sigma'}\left(\left[(f \wedge g) \vee (\overline{f} \wedge h)\right]_{x=0}\right) > \\
&\overset{2}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge h_{x=1}) \vee (\overline{x} \wedge [(f \wedge g) \vee (\overline{f} \wedge h)]_{x=0})\right) \\
&\overset{3}{=} && \mathcal{Z}_{\Sigma}\left((x \wedge [(0 \wedge g_{x=0}) \vee (1 \wedge h_{x=1})]) \vee (\overline{x} \wedge [(f \wedge g) \vee (\overline{f} \wedge h)]_{x=0})\right) \\
&\overset{4}{=} && \mathcal{Z}_{\Sigma}\left(\left(x \wedge \left[(f_{x=1} \wedge g_{x=1}) \vee (\overline{f_{x=1}} \wedge h_{x=1})\right]\right) \vee (\overline{x} \wedge [(f \wedge g) \vee (\overline{f} \wedge h)]_{x=0})\right) \\
&\overset{5}{=} && \mathcal{Z}_{\Sigma}\left(\left(x \wedge \left[(f \wedge g) \vee (\overline{f} \wedge h)\right]_{x=1}\right) \vee (\overline{x} \wedge [(f \wedge g) \vee (\overline{f} \wedge h)]_{x=0})\right) \\
&\overset{6}{=} && \mathcal{Z}_{\Sigma}\left((f \wedge g) \vee (\overline{f} \wedge h)\right)
\end{aligned}$$

   Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by logic equation. Step 4: as $A.t \neq x$, x has negative value in f (rule **R.4**). Step 5: by distribution of restriction. Step 6: by Shannon decomposition.

8. If $A.t = x, B.t = x, C.t \neq x$, then the function returns

$< x, \texttt{ITE}(A_{x=1}, B_{x=1}, 0'_{\Sigma}), \texttt{ITE}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < x, \mathcal{Z}_{\Sigma'}\left((f_{x=1} \wedge g_{x=1}) \vee (\overline{f_{x=1}} \wedge 0)\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{2}{=} < x, \mathcal{Z}_{\Sigma'}\left((f_{x=1} \wedge g_{x=1}) \vee (\overline{f_{x=1}} \wedge h_{x=1})\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{3}{=} < x, \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{4}{=} \mathcal{Z}_{\Sigma' \cup x}\left(\left(x \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right) \vee \left(\overline{x} \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right)\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma' \cup x}\left((f \wedge g) \vee (\overline{f} \wedge h)\right) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma}\left((f \wedge g) \vee (\overline{f} \wedge h)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: as $C.t \neq x$, x has negative value in h. Step 3: by distribution of restriction. Step 4: by definition of `CreateNode`. Step 5: by Shannon decomposition. Step6: by assumption A, B,and C are well-formed ZDDs.

9. If $A.t = x, B.t \neq x, C.t = x$, then the function returns

$< x, \texttt{ITE}(A_{x=1}, 0'_{\Sigma}, C_{x=1}), \texttt{ITE}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < x, \mathcal{Z}_{\Sigma'}\left((f_{x=1} \wedge 0) \vee (\overline{f_{x=1}} \wedge h_{x=1})\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{2}{=} < x, \mathcal{Z}_{\Sigma'}\left((f_{x=1} \wedge g_{x=1}) \vee (\overline{f_{x=1}} \wedge h_{x=1})\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{3}{=} < x, \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{4}{=} \mathcal{Z}_{\Sigma' \cup x}\left(\left(x \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right) \vee \left(\overline{x} \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right)\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma' \cup x}\left((f \wedge g) \vee (\overline{f} \wedge h)\right) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma}\left((f \wedge g) \vee (\overline{f} \wedge h)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: as $B.t \neq x$, x has negative value in g. Step 3: by distribution of restriction. Step 4: by definition of `CreateNode`. Step 5: by Shannon decomposition. Step6: by assumption A, B,and C are well-formed ZDDs.

10. If $A.t = x, B.t = x, C.t = x$, then the function returns

$< x, \texttt{ITE}_{x=1}, \texttt{ITE}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < x, \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right), \mathcal{Z}_{\Sigma'}\left(((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right) > \\
&\overset{2}{=} \mathcal{Z}_{\Sigma' \cup x}\left(\left(x \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=1}\right) \vee \left(\overline{x} \wedge ((f \wedge g) \vee (\overline{f} \wedge h))_{x=0}\right)\right) \\
&\overset{3}{=} \mathcal{Z}_{\Sigma' \cup x}\left((f \wedge g) \vee (\overline{f} \wedge h)\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma}\left((f \wedge g) \vee (\overline{f} \wedge h)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 5: by Shannon decomposition. Step6: by by assumption A, B,and C are well-formed ZDDs.

□

A.3 `Not` Operation

Definition A.5. Let v_Σ be shorthand for $\mathcal{Z}_\Sigma(v), v \in \{0,1\}$, x be the top variable in Σ and let $A = \mathcal{Z}_\Sigma(f)$. Then N is defined as follows:

$$N(A) = \begin{cases} 0_\Sigma & A = 1_\Sigma \\ 1_\Sigma & A = 0_\Sigma \\ < x, 1_{\Sigma'}, N(A_{x=0}, \Sigma') > & x \neq A.t \\ < x, N(A_{x=1}, \Sigma'), N(A_{x=0}, \Sigma') > & x = A.t \end{cases}$$

Theorem A.6. *The result of $N(\mathcal{Z}_\Sigma(f))$ is equal to $\mathcal{Z}_\Sigma(\overline{f})$.*

Proof. By induction on the size of Σ. We assume the induction hypothesis $N(A_{x=v}) \equiv \mathcal{Z}_{\Sigma'}(\overline{f_{x=v}})$, $A_{x=v} = \mathcal{Z}_{\Sigma'}(f_{x=v})$, where $\Sigma' = \Sigma \setminus \{x\}$. There are four cases:

1. If $A = 1_\Sigma$, then the function returns $0_\Sigma = \mathcal{Z}_\Sigma(0) = \mathcal{Z}_\Sigma(\overline{1}) = \mathcal{Z}_\Sigma(\overline{f})$, since $0 = \overline{1}$ and rules **R.2**.

2. If $A = 0_\Sigma$, then the function returns $1_\Sigma = \mathcal{Z}_\Sigma(1) = \mathcal{Z}_\Sigma(\overline{0}) = \mathcal{Z}_\Sigma(\overline{f})$, since $1 = \overline{0}$ and rules **R.1**.

3. If $x \neq A.t$, then the function returns $< x, 1_{\Sigma'}, N(A_x = 0, \Sigma') >$

$$\begin{aligned} &\overset{1}{=} < x, \mathcal{Z}_{\Sigma'}(1), N(A_x = 0, \Sigma') > \\ &\overset{2}{=} < x, \mathcal{Z}_{\Sigma'}(1), \mathcal{Z}_{\Sigma'}(\overline{f}) > \\ &\overset{3}{=} \mathcal{Z}_{\Sigma' \cup x}\left((x \wedge 1) \vee (\overline{x} \wedge \overline{f})\right) \\ &\overset{4}{=} \mathcal{Z}_\Sigma\left((x \wedge 1) \vee (\overline{x} \wedge \overline{f})\right) \\ &\overset{5}{=} \mathcal{Z}_\Sigma\left(x \vee (\overline{x} \wedge \overline{f})\right) \\ &\overset{6}{=} \mathcal{Z}_\Sigma\left(x \vee \overline{f}\right) \\ &\overset{7}{=} \mathcal{Z}_\Sigma\left(\overline{f}\right) \end{aligned}$$

Step 1: by definition of 1_Σ. Step 2: by induction hypothesis. Step 3: by definition of `CreateNode`. Step 4: by assumption A is a well-formed ZDD. Step 5,6: by logical equivalence. Step 7: as $x < A.t$, x has negative value in f(rule **R.4**).

4. If $x = A.t$, then the function returns $< x, N(A_{x=1}, \Sigma'), N(A_{x=0}, \Sigma') >$

$$
\begin{aligned}
&\overset{1}{=} < x, \mathcal{Z}_{\Sigma'}\left(\overline{f_{x=1}}\right), \mathcal{Z}_{\Sigma'}\left(\overline{f_{x=0}}\right) > \\
&\overset{2}{=} \mathcal{Z}_{\Sigma' \cup x}\left((x \wedge \overline{f_{x=1}}) \vee (\overline{x} \wedge \overline{f_{x=0}})\right) \\
&\overset{3}{=} \mathcal{Z}_{\Sigma}\left((x \wedge \overline{f_{x=1}}) \vee (\overline{x} \wedge \overline{f_{x=0}})\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma}\left((x \vee \overline{f_{x=1}}) \wedge (\overline{x} \vee \overline{f_{x=0}})\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma}\left(\overline{(x \wedge f_{x=1})} \wedge \overline{(\overline{x} \wedge f_{x=0})}\right) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma}\left(\overline{(x \wedge f_{x=1}) \vee (\overline{x} \wedge f_{x=0})}\right) \\
&\overset{7}{=} \mathcal{Z}_{\Sigma}\left(\overline{f}\right)
\end{aligned}
$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by assumption A is a well-formed ZDD. Step 4: by Shannon expansion rule. Step 5,6: by logical equivalent. Step 7: by Shannon decomposition

□

A.4 `Exist` Operation

Definition A.7. Let $A = \mathcal{Z}_\Sigma(f)$ be a well-formed ZDD, v_Σ be shorthand for $\mathcal{Z}_\Sigma(v)$ and let $\texttt{EX}_{x=v}$ be shorthand for $\texttt{EX}(A_{x=v}, \Sigma', X)$, where $v \in \{0,1\}$. Also let x be the top variable in domain Σ. Then $\texttt{EX}$ will be defined as follows:

$$\texttt{EX}(A, \Sigma, X) = \begin{cases} 0_{\Sigma - X} & A = 0_\Sigma \\ 1_{\Sigma - X} & A = 1_\Sigma \\ \texttt{ITE}(\texttt{EX}_{x=1}, 1_{\Sigma'-X}, \texttt{EX}_{x=0}, \Sigma' - X) & A.t = x \wedge x \in X \\ < x, \texttt{EX}_{x=1}, \texttt{EX}_{x=0} > & A.t = x \wedge x \notin X \\ < x, 0, \texttt{EX}_{x=0} > & A.t \neq x \wedge x \notin X \\ \texttt{EX}_{x=0} & A.t \neq x \wedge x \in X \end{cases}$$

Theorem A.8. *The result of* $\texttt{EX}(\mathcal{Z}_\Sigma(f), X)$ *is equal to* $\mathcal{Z}_{\Sigma-X}(\exists X.f)$.

Proof. By induction on the size of Σ. We assume the induction hypothesis $\texttt{EX}(\mathcal{Z}_{\Sigma'}(f_{x=v}), X) \equiv \mathcal{Z}_{\Sigma-X}(\exists X.f_{x=v})$. There are five cases:

1. If $A = 0$, then the function returns $0 = \mathcal{Z}_{\Sigma-X}(0) = \mathcal{Z}_{\Sigma-X}(\exists X.0) = \mathcal{Z}_{\Sigma-X}(\exists X.f)$, by rule **R.1**.

2. If $A = 1_\Sigma$, then the function returns $1_{\Sigma-X} = \mathcal{Z}_{\Sigma-X}(1) = \mathcal{Z}_{\Sigma-X}(\exists X.1) = \mathcal{Z}_{\Sigma-X}(\exists X.f)$, by rule **R.2**.

3. If $A.t = x \wedge x \in X$, then the function returns $\texttt{ITE}(\texttt{EX}_{x=1}, 1_{\Sigma'-X}, \texttt{EX}_{x=0}, \Sigma' - X)$

$$\begin{aligned} &\overset{1}{=} \texttt{ITE}(\mathcal{Z}_{\Sigma'-X}(\exists X.f_{x=1}), \mathcal{Z}_{\Sigma'-X}(1), \mathcal{Z}_{\Sigma'-X}(\exists X.f_{x=0}), \Sigma' - X) \\ &\overset{2}{=} \mathcal{Z}_{\Sigma'-X}\left((\exists X.f_{x=1} \wedge 1) \vee (\overline{\exists X.f_{x=1}} \wedge \exists X.f_{x=0})\right) \\ &\overset{3}{=} \mathcal{Z}_{\Sigma'-X}(\exists X.(f_{x=1}) \vee \exists X.(f_{x=0}))) \\ &\overset{4}{=} \mathcal{Z}_{\Sigma'-X}(\exists X.(f_{x=0} \vee f_{x=1})) \\ &\overset{5}{=} \mathcal{Z}_{\Sigma'-X}(\exists X \exists x.f) \\ &\overset{6}{=} \mathcal{Z}_{\Sigma'-X}(\exists X.f) \\ &\overset{7}{=} \mathcal{Z}_{\Sigma-X}(\exists X.f) \end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `ITE`. Step 3: by logical equation. Step 4: by distribution of $\exists$ over $\vee$. Step 5: by definition of $\exists$. Step 6,7: as $x \in X$.

4. If $A.t = x \wedge x \notin X$, then the function returns $< x, \mathtt{EX}_{x=1}, \mathtt{EX}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < x, \mathcal{Z}_{\Sigma'-X}\left(\exists X.f_{x=1}\right), \mathcal{Z}_{\Sigma'-X}\left(\exists X.f_{x=0}\right) > \\
&\overset{2}{=} \mathcal{Z}_{(\Sigma'-X)\cup x}\left((x \wedge \exists X.f_{x=1}) \vee (\overline{x} \wedge \exists X.f_{x=0})\right) \\
&\overset{3}{=} \mathcal{Z}_{(\Sigma-X)}\left((x \wedge \exists X.f_{x=1}) \vee (\overline{x} \wedge \exists X.f_{x=0})\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma-X}\left((\exists X.(x \wedge f_{x=1})) \vee (\exists X.(\overline{x} \wedge f_{x=0}))\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.((x \wedge f_{x=1}) \vee (\overline{x} \wedge f_{x=0}))\right) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.f\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by distribution $\cup$ over $-$ and since $x \notin X$ Step 4: as $x \notin X$. Step 5: by distribution of $\exists$ over $\vee$. Step 6: by Shannon decomposition.

5. If $A.t \neq x \wedge x \notin X$, then the function returns $< x, 0, \mathtt{EX}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < x, 0, \mathcal{Z}_{\Sigma'-X}\left(\exists X.f_{x=0}\right) > \\
&\overset{2}{=} \mathcal{Z}_{(\Sigma'-X)\cup x}\left((x \wedge 0) \vee (\overline{x} \wedge \exists X.f_{x=0})\right) \\
&\overset{3}{=} \mathcal{Z}_{\Sigma-X}\left((x \wedge 0) \vee (\overline{x} \wedge \exists X.f_{x=0}))\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma-X}\left((x \wedge \exists X.f_{x=1}) \vee (\overline{x} \wedge \exists X.f_{x=0}))\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.f)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by distribution $\cup$ over $-$ and since $x \notin X$. Step 4: by rule **R.4**. Step 5: by Shannon decomposition.

6. If $A.t \neq x \wedge x \in X$, then the function returns $\mathtt{EX}_{x=0}$

$$\begin{aligned}
&\overset{1}{=} \mathcal{Z}_{\Sigma'-X}\left(\exists X.f_{x=0}\right) \\
&\overset{2}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.f_{x=0}\right) \\
&\overset{3}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.(f_{x=0} \vee 0)\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.(f_{x=0} \vee f_{x=1})\right) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.\exists x.f\right) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma-X}\left(\exists X.f\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: as $x \in X$. Step 3: by logical equivalence. Step 4: by rule **R.4**. Step 5: by definition of $\exists$. Step 6: as $x \in X$

$\square$

A.5 `Rename` Operation

Definition A.9. Let $A = \mathcal{Z}_{\Sigma}(f)$ be a well-formed ZDD, where x is the top level of Σ, and let y be the top variable of A. We also assume that substituting x with x' does not change the ordering. Then `R` will be defined as follows:

$$\texttt{R}(A, x, x') = \begin{cases} \mathcal{Z}_{(\Sigma - x)\cup x'}(f[x/x']) & A = 0 \vee A = 1 \\ < y, \texttt{R}(A_{y=1}, x, x'), \texttt{R}(A_{y=0}, x, x') > & y < x \\ < x', A_{x=1}, A_{x=0} > & y = x \\ < x', 0, A_{x=0} > & y > x \end{cases}$$

Theorem A.10. *The result of* $\texttt{R}(\mathcal{Z}_{\Sigma}(f), x, x')$ *is equal to* $\mathcal{Z}_{(\Sigma - x)\cup x'}(f[x/x'])$.

Proof. By induction on the size of Σ. We assume the induction hypothesis $\texttt{R}(\mathcal{Z}_{\Sigma}(f_{x=v}), x, x') \equiv \mathcal{Z}_{(\Sigma - x)\cup x'}(f_{x=v}[x/x'])$. There are four cases:

1. If $A = 0$ Or $A = 1$, then $\mathcal{Z}_{(\Sigma - x)\cup x'}(f[x/x'])$
2. If $y < x$, then $< y, \texttt{R}(A_{y=1}, x, x'), \texttt{R}(A_{y=0}, x, x') >$

$$\begin{aligned} &\overset{1}{=} < y, \mathcal{Z}_{(\Sigma - y - x)\cup x'}\left(f_{y=1}[x/x']\right), \mathcal{Z}_{(\Sigma - y - x)\cup x'}\left(f_{y=0}[x/x']\right) > \\ &\overset{2}{=} \mathcal{Z}_{(\Sigma - y - x)\cup x' \cup y}\left((y \wedge f_{y=1}[x/x']) \vee (\overline{y} \wedge f_{y=0}[x/x'])\right) \\ &\overset{3}{=} \mathcal{Z}_{(\Sigma - x)\cup x'}\left((y \wedge f_{y=1}[x/x']) \vee (\overline{y} \wedge f_{y=0}[x/x'])\right) \\ &\overset{4}{=} \mathcal{Z}_{(\Sigma - x)\cup x'}\left(f[x/x']\right) \end{aligned}$$

   Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by assumption,A is a well-formed ZDD. Step 4: by Shannon decomposition.

3. If $y = x$, then $< x', A_{x=1}, A_{x=0} >$

$$\begin{aligned} &\overset{1}{=} < x', \mathcal{Z}_{(\Sigma - x)}(f_{x=1}), \mathcal{Z}_{\Sigma - x}(f_{x=0}) > \\ &\overset{2}{=} \mathcal{Z}_{(\Sigma - x)\cup x'}\left((x' \wedge f_{x=1}) \vee (\overline{x'} \wedge f_{x=0})\right) \\ &\overset{3}{=} \mathcal{Z}_{(\Sigma - x)\cup x'}\left((x' \wedge f_{x'=1}[x/x']) \vee (\overline{x'} \wedge f_{x'=0}[x/x'])\right) \\ &\overset{4}{=} \mathcal{Z}_{(\Sigma - x)\cup x'}\left(f[x/x']\right) \end{aligned}$$

   Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3:as x doesn't appear in $f_{x=v}$, by renaming x with x' it does not change . Step 4: by Shannon decomposition.

4. If $y > x$, then $< x', 0, A_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} \quad < x', 0, \mathcal{Z}_{\Sigma - x}\left(f_{x=0}\right) > \\
&\overset{2}{=} \quad \mathcal{Z}_{(\Sigma - x)\cup x'}\left((x' \wedge 0) \vee (\overline{x'} \wedge f_{x=0})\right) \\
&\overset{3}{=} \quad \mathcal{Z}_{(\Sigma - x)\cup x'}\left((x' \wedge f_{x=1}) \vee (\overline{x'} \wedge f_{x=0})\right) \\
&\overset{4}{=} \quad \mathcal{Z}_{(\Sigma - x)\cup x'}\left((x' \wedge f_{x'=1}[x/x']) \vee (\overline{x'} \wedge f_{x'=0}[x/x'])\right) \\
&\overset{5}{=} \quad \mathcal{Z}_{(\Sigma - x)\cup x'}\left(f[x/x']\right)
\end{aligned}$$

Step 1: by definition. Step 2: by definition of `CreateNode`. Step 3: as $A.t > x$, x has negative value in f(rule **R.4**). Step 4: as x doesn't appear in $f_{x=v}$, by renaming x with x' it does not change . Step 5: by Shannon decomposition.

□

A.6 `RelProd` Operation

Definition A.11. Let $A = \mathcal{Z}_{\Sigma}(f)$ and $B = \mathcal{Z}_{\Sigma}(h)$ be well-formed ZDDs, $\texttt{RP}_{x=v}$ be the shorthand for $\texttt{RP}(A_{x=v}, B_{x=v}, \Sigma', X)$, and v_{Σ} be the shorthand for $\mathcal{Z}_{\Sigma}(v), v \in \{0,1\}$. x be the top variable in domain Σ. Then the definition of RP is as follows:

$$\texttt{RP}(A,B,\Sigma,X) = \begin{cases} 1_{\Sigma-X} & A = 1_{\Sigma}, B = 1_{\Sigma} \\ 0_{\Sigma-X} & A = 0 \vee B = 0 \\ \texttt{ITE}(\texttt{RP}_{x=0}, \mathcal{Z}_{\Sigma-X}(1), \texttt{RP}_{x=1}) & x \in X \\ < x, \texttt{RP}_{x=1}, \texttt{RP}_{x=0} > & x \notin X, x = A.t, x = B.t \\ < x, 0, \texttt{RP}_{x=0} > & \text{Otherwise} \end{cases}$$

Note. In `RelProd` definition, it is assumed that A and B have the same domain Σ.

Theorem A.12. *The result of* $\texttt{RP}(\mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma}(h), X)$ *is equal to* $\mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))$.

Proof. By induction on the size of Σ. We assume the induction hypothesis $\texttt{RP}(\mathcal{Z}_{\Sigma}(f_{x=v}), \mathcal{Z}_{\Sigma}(h_{x=v}), X) \equiv \mathcal{Z}_{\Sigma-X-x}(\exists X.(f_{x=v} \wedge h_{x=v})) \equiv \mathcal{Z}_{\Sigma-X-x}(\exists X.(f \wedge h)_{x=v})$, by distribution of restriction over $\wedge$. There are five cases:

1. If $A = 1_{\Sigma}$ and $B = 1_{\Sigma}$, then the function returns $\mathcal{Z}_{\Sigma-X}(1) = \mathcal{Z}_{\Sigma-X}(1 \wedge 1) = \mathcal{Z}_{\Sigma-X}(\exists X.(1 \wedge 1)) = \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))$, by rule **R.2**.

2. If $A = 0$, then the function returns $\mathcal{Z}_{\Sigma-X}(0) = \mathcal{Z}_{\Sigma-X}(\exists X.0) = \mathcal{Z}_{\Sigma-X}(\exists X.(0 \wedge h)) = \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))$, by rule **R.1**.

   If $B = 0$, then the function returns $\mathcal{Z}_{\Sigma-X}(0) = \mathcal{Z}_{\Sigma-X}(\exists X.0) = \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge 0)) = \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))$, by rule **R.1**.

3. If $x \in X$, then the function returns $\texttt{ITE}(\texttt{RP}_{x=0}, \mathcal{Z}_{\Sigma-X-x}(1), \texttt{RP}_{x=1})$

$$\begin{aligned}
&\overset{1}{=} \texttt{ITE}(\mathcal{Z}_{\Sigma-X-x}(\exists X.(f \wedge h)_{x=0}), \mathcal{Z}_{\Sigma-X}(1), \mathcal{Z}_{\Sigma-X-x}(\exists X.(f \wedge h)_{x=1})) \\
&\overset{2}{=} \texttt{ITE}(\mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h)_{x=0}), \mathcal{Z}_{\Sigma-X}(1), \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h)_{x=1})) \\
&\overset{3}{=} \mathcal{Z}_{\Sigma-X}\left((\exists X.(f \wedge h)_{x=0} \wedge 1) \vee \left(\overline{\exists X.(f \wedge h)_{x=0}} \wedge \exists X.(f \wedge h)_{x=1}\right)\right) \\
&\overset{4}{=} \mathcal{Z}_{\Sigma-X}((\exists X.(f \wedge h)_{x=0}) \vee (\exists X.(f \wedge h)_{x=1})) \\
&\overset{5}{=} \mathcal{Z}_{\Sigma-X}(\exists X.((f \wedge h)_{x=0} \vee (f \wedge h)_{x=1})) \\
&\overset{6}{=} \mathcal{Z}_{\Sigma-X}(\exists X.\exists x.(f \wedge h)) \\
&\overset{7}{=} \mathcal{Z}_{\Sigma-X}(\exists X.(f \wedge h))
\end{aligned}$$

Step 1: by induction hypothesis. Step 2 : as $x \in X$. Step 3: by definition of `ITE`. Step 4: by logical equation. Step 5: by distribution of $\exists$ over $\vee$. Step 6: by definition of $\exists$. Step 7: as $x \in X$.

4. If $x \notin X$ and $x = A.t$ and $x = B.t$, then the function returns $< x, \texttt{RP}_{x=1}, \texttt{RP}_{x=0} >$

$$
\begin{aligned}
&\overset{1}{=} \; < x, \mathcal{Z}_{\Sigma'-X-x}\left(\exists X.\left(f \wedge h\right)_{x=1}\right), \mathcal{Z}_{\Sigma'-X-x}\left(\exists X.\left(f \wedge h\right)_{x=0}\right) > \\
&\overset{2}{=} \; \mathcal{Z}_{(\Sigma'-X-x)\cup x}\left((x \wedge \exists X.(f \wedge h)_{x=1}) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) \\
&\overset{3}{=} \; \mathcal{Z}_{(\Sigma'-X)\cup x}\left((x \wedge \exists X.(f \wedge h)_{x=1}) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) \\
&\overset{4}{=} \; \mathcal{Z}_{\Sigma-X}\left((x \wedge \exists X.(f \wedge h)_{x=1}) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) \\
&\overset{5}{=} \; \mathcal{Z}_{\Sigma-X}\left(\exists X.\,(x \wedge (f \wedge h)_{x=1}) \vee \exists X.\,(\overline{x} \wedge (f \wedge h)_{x=0})\right) \\
&\overset{6}{=} \; \mathcal{Z}_{\Sigma-X}\left(\exists X.\,((x \wedge (f \wedge h)_{x=1}) \vee (\overline{x} \wedge (f \wedge h)_{x=0}))\right) \\
&\overset{7}{=} \; \mathcal{Z}_{\Sigma-X}\left(\exists X.(f \wedge h)\right)
\end{aligned}
$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: as $x \notin \Sigma'$. Step 4: by distribution $\cup$ over $-$ and since $x \notin X$. Step 5: as $x \notin X$. Step 6: by distribution of $\exists$ over $\vee$. Step 7: by Shannon decomposition.

5. If $x \notin X$ and $(x \neq A.t \vee x \neq B.t)$, then the function returns $< x, 0, \texttt{RP}_{x=0} >$

$$
\begin{aligned}
&\overset{1}{=} \; < x, 0, \mathcal{Z}_{\Sigma'-X-x}\left(\exists X.\left(f \wedge h\right)_{x=0}\right) > \\
&\overset{2}{=} \; \mathcal{Z}_{(\Sigma'-X-x)\cup x}\left((x \wedge 0) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) > \\
&\overset{3}{=} \; \mathcal{Z}_{(\Sigma'-X)\cup x}\left((x \wedge 0) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) > \\
&\overset{4}{=} \; \mathcal{Z}_{(\Sigma-X)}\left((x \wedge 0) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) > \\
&\overset{5}{=} \; \mathcal{Z}_{(\Sigma-X)}\left((x \wedge \exists X.(f_{x=1} \wedge h_{x=1})) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) > \\
&\overset{6}{=} \; \mathcal{Z}_{(\Sigma-X)}\left((x \wedge \exists X.(f \wedge h)_{x=1}) \vee (\overline{x} \wedge \exists X.(f \wedge h)_{x=0})\right) > \\
&\overset{7}{=} \; \mathcal{Z}_{(\Sigma-X)}\left(\exists X.(x \wedge (f \wedge h)_{x=1}) \vee \exists X.(\overline{x} \wedge (f \wedge h)_{x=0})\right) > \\
&\overset{8}{=} \; \mathcal{Z}_{(\Sigma-X)}\left(\exists X.\,((x \wedge (f \wedge h)_{x=1}) \vee (\overline{x} \wedge (f \wedge h)_{x=0}))\right) > \\
&\overset{9}{=} \; \mathcal{Z}_{\Sigma-X}\left(\exists X.(f \wedge h)\right)
\end{aligned}
$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: as $x \notin \Sigma'$. Step 4: by distribution $\cup$ over $-$ and since $x \notin X$. Step 5: if $x \neq A.t$, then $f_{x=1}$ is 0. And if $x \neq B.t$, then $h_{x=1}$ is 0(rule **R.4**). Step 6: by distribution of restriction. Step 7: as $x \notin X$. Step 8: by distribution of $\exists$ over $\vee$. Step 9: by Shannon decomposition.

$\square$

A.7 RelProdS Operation

Definition A.13. Let $A = \mathcal{Z}_{\Sigma}(f)$, $B = \mathcal{Z}_{\Sigma'}(h)$ be well-formed ZDDs, and $\texttt{RPS}_{x=v}$ be the shorthand for $\texttt{RPS}(A_{x=v}, B_{x=v}, X, S)$, v_{Σ} be the shorthand for $\mathcal{Z}_{\Sigma}(v), v \in \{0,1\}$, and $x = max(\Sigma.t, \Sigma'.t)$. It is also assumed that renaming each x with $S(x)$ will not change the ordering. Then the definition of RPS is as follows:

$$\texttt{RPS}(A,B,X,S) = \begin{cases} 1_{(\Sigma\cup\Sigma'-X)[S]} & A = 1_{\Sigma}, B = 1_{\Sigma'} \\ 0_{(\Sigma\cup\Sigma'-X)[S]} & A = 0_{\Sigma} \vee B = 0_{\Sigma'} \\ \texttt{ITE}(\texttt{RPS}_{x=0}, 1_{(\Sigma\cup\Sigma'-X-x)[S]}, \texttt{RPS}_{x=1}) & x \in X \\ < S(x), \texttt{RPS}_{x=1}, \texttt{RPS}_{x=0} > & \text{Otherwise} \end{cases}$$

Note. In RelProdS definition, A and B can have the different domains Σ and Σ', respectively.

Theorem A.14. *The result of RPS*$(\mathcal{Z}_{\Sigma}(f), \mathcal{Z}_{\Sigma'}(h), X, S)$ *is equal to* $\mathcal{Z}_{(\Sigma\cup\Sigma'-X)[S]}(\exists X.(f \wedge h)[S])$.

Proof. By induction on the size of Σ and Σ', we assume the induction hypothesis
$\texttt{RPS}(\mathcal{Z}_{\Sigma}(f_{x=v}), \mathcal{Z}_{\Sigma'}(h_{x=v}), X, S) \equiv \mathcal{Z}_{(\Sigma\cup\Sigma'-X-x)[S]}(\exists X.(f_{x=v} \wedge h_{x=v}))$
$\equiv \mathcal{Z}_{(\Sigma\cup\Sigma'-X-x)[S]}(\exists X.(f \wedge h)_{x=v})$ (by distribution of restriction over $\wedge$).

Also let Σ'' and $\Sigma'' - x$ be shorthand for $(\Sigma \cup \Sigma' - X)[S]$ and $(\Sigma \cup \Sigma' - X - x)[S]$, respectively. There are four cases:

1. If $A = 1_{\Sigma}$ And $B = 1_{\Sigma'}$, then the function returns $1_{\Sigma''} = \mathcal{Z}_{\Sigma''}(1) = \mathcal{Z}_{\Sigma''}(1 \wedge 1) = \mathcal{Z}_{\Sigma''}(\exists X.(1 \wedge 1)[S]) = \mathcal{Z}_{\Sigma''}(\exists X.(f \wedge h)[S])$, by rule **R.2**.

2. If $A = 0_{\Sigma}$, then the function returns $0_{\Sigma''} = \mathcal{Z}_{\Sigma''}(0) = \mathcal{Z}_{\Sigma''}(\exists X.0[S]) = \mathcal{Z}_{\Sigma''}(\exists X.(0 \wedge h)[S]) = \mathcal{Z}_{\Sigma''}(\exists X.(f \wedge h)[S])$, by rule **R.1**.

   If $B = 0_{\Sigma'}$, then the function returns $0_{\Sigma''} = \mathcal{Z}_{\Sigma''}(0) = \mathcal{Z}_{\Sigma''}(\exists X.0[S]) = \mathcal{Z}_{\Sigma''}(\exists X.(f \wedge 0)[S]) = \mathcal{Z}_{\Sigma''}(\exists X.(f \wedge h)[S])$, by rule **R.1**.

3. If $x \in X$, then the function returns $\mathtt{ITE}(\mathtt{RPS}_{x=0}, 1_{\Sigma''-x}, \mathtt{RPS}_{x=1})$

$$\begin{aligned}
&\overset{1}{=} \mathtt{ITE}(\mathcal{Z}_{\Sigma''-x}\left(\exists X.(f \wedge h)_{x=0}[S]\right), \mathcal{Z}_{\Sigma''-x}\left(1\right), \mathcal{Z}_{\Sigma''-x}\left(\exists X.(f \wedge h)_{x=1}[S]\right))\\
&\overset{2}{=} \mathcal{Z}_{\Sigma''-x}\left((\exists X.(f \wedge h)_{x=0}[S] \wedge 1) \vee \left(\overline{\exists X.(f \wedge h)_{x=0}[S]} \wedge \exists X.(f \wedge h)_{x=1}[S]\right)\right)\\
&\overset{3}{=} \mathcal{Z}_{\Sigma''}\left((\exists X.(f \wedge h)_{x=0}[S] \wedge 1) \vee \left(\overline{\exists X.(f \wedge h)_{x=0}[S]} \wedge \exists X.(f \wedge h)_{x=1}[S]\right)\right)\\
&\overset{4}{=} \mathcal{Z}_{\Sigma''}\left((\exists X.(f \wedge h)_{x=0}[S]) \vee (\exists X.(f \wedge h)_{x=1}[S])\right)\\
&\overset{5}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.\left((f \wedge h)_{x=0}[S] \vee (f \wedge h)_{x=1}[S]\right)\right)\\
&\overset{6}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.\exists x.(f \wedge h)[S]\right)\\
&\overset{7}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.(f \wedge h)[S]\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `ITE`. Step 3: as $x \in X$. Step 4: by logical equation. Step 5: by distribution of $\exists$ over $\vee$. Step 6: by definition of $\exists$. Step 7: as $x \in X$.

4. If $x \notin X$, then the function returns $< S(x), \mathtt{RPS}_{x=1}, \mathtt{RPS}_{x=0} >$

$$\begin{aligned}
&\overset{1}{=} < S(x), \mathcal{Z}_{\Sigma''-x}\left(\exists X.(f \wedge h)_{x=1}\right), \mathcal{Z}_{\Sigma''-x}\left(\exists X.(f \wedge h)_{x=0}\right) >\\
&\overset{2}{=} \mathcal{Z}_{(\Sigma \cup \Sigma' - X - x)[S] \cup S(x)}\left((S(x) \wedge 0) \vee (\overline{S(x)} \wedge \exists X.(f \wedge h)_{x=0})\right)\\
&\overset{3}{=} \mathcal{Z}_{(\Sigma \cup \Sigma' - X)[S]}\left((S(x) \wedge 0) \vee (\overline{S(x)} \wedge \exists X.(f \wedge h)_{x=0})\right)\\
&\overset{4}{=} \mathcal{Z}_{\Sigma''}\left((S(x) \wedge \exists X.(f_{x=1} \wedge h_{x=1})) \vee (\overline{S(x)} \wedge \exists X.(f \wedge h)_{x=0})\right)\\
&\overset{5}{=} \mathcal{Z}_{\Sigma''}\left((S(x) \wedge \exists X.(f \wedge h)_{x=1}) \vee (\overline{S(x)} \wedge \exists X.(f \wedge h)_{x=0})\right)\\
&\overset{6}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.(S(x) \wedge (f \wedge h)_{x=1}) \vee \exists X.(\overline{S(x)} \wedge (f \wedge h)_{x=0})\right)\\
&\overset{7}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.\left((S(x) \wedge (f \wedge h)_{x=1}) \vee (\overline{S(x)} \wedge (f \wedge h)_{x=0})\right)\right)\\
&\overset{8}{=} \mathcal{Z}_{\Sigma''}\left(\exists X.(f \wedge h)\right)
\end{aligned}$$

Step 1: by induction hypothesis. Step 2: by definition of `CreateNode`. Step 3: by substitution definition. Step 4: if $x \neq A.t$, then $f_{x=1}$ is 0 (rule **R.4**). And if $x \neq B.t$, then $h_{x=1}$ is 0. Step 5: by distribution of restriction. Step 6: as $x \notin X$. Step 7: by distribution of $\exists$ over $\vee$. Step 8: by Shannon decomposition.

□

Bibliography

[1] BEEM database. http://anna.fi.muni.cz/models/.

[2] gperftools: A set of performance tools for c++ developers. https://code.google.com/p/gperftools/.

[3] S.B. Akers. Binary decision diagrams. volume C-27, pages 509–516. IEEE Computer Society, June 1978.

[4] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications, 1993.

[5] V. Bertacco, S. Minato, P. Verplaetse, L. Benini, and G. De Micheli. Decision diagrams and pass transistor logic synthesis. In *In Int'l Workshop on Logic Synth*, Stanford, CA, USA, 1997. Stanford University.

[6] Stefan Blom, Jaco van de Pol, and Michael Weber. Ltsmin: Distributed and symbolic reachability. In T. Touili, B. Cook, and P. Jackson, editors, *Computer Aided Verification*, volume 6174 of *Lecture Notes in Computer Science*, pages 354–359, Berlin, Germany, July 2010. Springer Verlag. See also Technical Report TR-CTIT-09-30 (http://eprints.eemcs.utwente.nl/15703/).

[7] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. volume 24, pages 293–318, New York, NY, USA, September 1992. ACM.

[8] O. Coudert. Two-level logic minimization: an overview. volume 17, pages 97 – 140. Elsevier Science Publishers B. V., 1994.

[9] O. Coudert. Solving graph optimization problems with ZBDDs. In *European Design and Test Conference, 1997. ED TC 97. Proceedings*, pages 224–228. IEEE Computer Society, 1997.

[10] O. Coudert. A new paradigm for dichotomy-based constrained encoding. In *Design, Automation and Test in Europe, 1998., Proceedings*, pages 830–834. IEEE Computer Society, 1998.

[11] O. Coudert and J. C. Madre. Implicit and incremental computation of primes and essential primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Design Automation Conference*, DAC '92, pages 36–39, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[12] O. Coudert and C.-J.R. Shi. Exact dichotomy-based constrained encoding. In *Computer Design: VLSI in Computers and Processors, 1996. ICCD '96. Proceedings., 1996 IEEE International Conference on*, pages 426–431, 1996.

[13] O. Coudert and C.-J.R. Shi. Exact multi-layer topological planar routing. In *Custom Integrated Circuits Conference, 1996., Proceedings of the IEEE 1996*, pages 179–182, 1996.

[14] R. Drechsler and D. Sieling. Special section on BDD binary decision diagrams in theory and practice. pages 112–136. Springer-Verlag, 2001.

[15] S. Ishihara and S. Minato. Manipulation of regular expressions under length constraints using Zero-suppressed BDDs. In *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal*, pages 391–396. IEEE Computer Society, 1995.

[16] J. Jacob and A. Mishchenko. Unate decomposition of boolean functions. In *PROC. IWLS '01*, pages 66–71, 2001.

[17] J. Lind-Nielsen. Buddy: A binary decision diagram library. `http://sourceforge.net/projects/buddy/`.

[18] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[19] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Design Automation, 1993. 30th Conference on*, pages 272–277. IEEE Computer Society, June 1993.

[20] S. Minato. Calculation of unate cube set algebra using Zero-suppressed BDDs. In *Design Automation, 1994. 31st Conference on*, pages 420–424. IEEE Computer Society, 1994.

[21] S. Minato. Implicit manipulation of polynomials using Zero-suppressed BDDs. In *European Design and Test Conference, 1995. ED TC 1995, Proceedings.*, pages 449–454. IEEE Computer Society, 1995.

[22] S. Minato. Zero-suppressed BDDs and their applications. volume 3, pages 156–170. Springer-Verlag, 2001.

[23] S. Minato and G. De Micheli. Finding all simple disjunctive decompositions using irredundant sum-of-products forms. In *ICCAD'98*, pages 111–117. IEEE, 1998.

[24] A. Mishchenko. Extra v. 2.0: Software library extending cudd package: Release 2.3.1. `http://web.cecs.pdx.edu/~alanmi/research/extra.htm`.

[25] A. Mishchenko. An introduction to zero-suppressed binary decision diagrams. Technical report, in ‘Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, 2001.

[26] R. Pelánek. Beem: Benchmarks for explicit model checkers. In *Proceedings of the 14th International SPIN Conference on Model Checking Software*, pages 263–267, Berlin, Heidelberg, 2007. Springer-Verlag.

[27] F. Somenzi. Cudd : Colorado university decision diagram package, release 2.5.0. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[28] M. Tomisaka and T. YONEDA. Partial order reduction in symbolic state space traversal using ZBDDs. *IEICE Trans. Fundamentals*, pages 1–8, 1999.

[29] T. van Dijk. The parallelization of binary decision diagram operations for model checking. `http://essay.utwente.nl/61650/`, 2012.

[30] T. van Dijk, A.W. Laarman, and J.C. van de Pol. Multi-core BDD operations for symbolic reachability. In K. Heljanko and W.J. Knottenbelt, editors, *11th International Workshop on Parallel and Distributed Methods in verifiCation, PDMC 2012*, Electronic Proceedings in Theoretical Computer Science. eptcs.org, September 2012.

[31] T. Yoneda, H. Hatori, A. Takahara, and S. Minato. BDDs vs. Zero-suppressed BDDs: for CTL symbolic model checking of Petri nets. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 435–449. Springer Berlin Heidelberg, 1996.