Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Symbolic Model Checking of Data-Intensive Systems

Bachelor's Thesis

Author
Adrián Soltész

Advisor
Vince Molnár
András Vörös

2015. december 11.

Contents

Kivonat **5**

Abstract **6**

1 Introduction **7**

2 Background **9**

2.1 Petri Nets . . . 9

2.2 Kripke Structures . . . 10

2.2.1 Kripke Structures as State Space Representations . . . 11

2.3 Model Checking . . . 12

2.3.1 Symbolic Model Checking . . . 13

2.4 Decision Diagrams . . . 13

2.4.1 Binary Decision Diagrams . . . 13

2.4.2 Multivalued Decision Diagrams . . . 13

2.4.2.1 Operations on Multivalued Decision Diagrams . . . 15

2.5 Algorithms for State Space Exploration . . . 16

2.5.1 Basic Algorithms . . . 16

2.5.2 Saturation . . . 16

3 Hierarchical Set Decision Diagrams **18**

3.1 Definition of Set Decision Diagrams . . . 18

3.2 Properties of Set Decision Diagrams . . . 18

3.3 Operations on Set Decision Diagrams . . . 20

3.3.1 Set Operations . . . 20

3.3.1.1 Intersection . . . 21

3.3.1.2 Union . . . 21

3.3.1.3 Subtraction . . . 22

4 Model Checking with Set Decision Diagrams **24**

4.1 Symbolic Representations . . . 24

4.1.1 Representation of the State Space . . . 24

4.1.1.1 Encoding States with Multivalued Decision Diagrams . . . 25

4.1.1.2 Ecoding States with Set Decision Diagrams . . . 25

4.1.2 Encoding of Petri Net Transitions . . . 26

4.1.2.1 Common Methods for Encoding Firing Rules . . . 27

4.1.2.2 Encoding Firing Rules of Hierarchical Models . . . 29

4.2 State Space Exploration with Set Desicison Diagrams . . . 30

4.2.1 Firing Transitions in Hierarchical Systems . . . 31

5 Implementation **33**
5.1 Overview of the Implemented Set Decision Diagram 33
5.2 Optimizations 35

6 Evaluation of the Results **36**
6.1 Results 36
6.1.1 Evaluation of the Measures 38

7 Conclusion and Future Work **39**
7.1 Contributions 39
7.2 Conclusion 39
7.3 Future Work 39

Acknowledgements **41**

List of Figures **42**

List of Tables **43**

List of Algorithms **44**

Bibliography **46**

Appendices **47**

A Description of Tested Models **48**

HALLGATÓI NYILATKOZAT

Alulírott *Soltész Adrián*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2015. december 11.

Soltész Adrián
hallgató

Kivonat

Biztonságkritikus rendszereknél egyre elterjedtebb a tervezésidejű verifikáció. Egy ilyen technika az ún. modellellenőrzés. Ahogy a rendszerek egyre komplexebbé válnak, úgy nő a rendszerek állapotváltozóinak száma is. Az ilyen „adatintenzív” rendszerek állapotainak száma robbanásszerűen nőhet, az állapotteret explicit módon már nem lehetséges eltárolni.

Ezen probléma megoldására jelentek meg a szimbolikus modellellenőrző technikák, melyek az állapotteret kompakt adatszerkezetek, jellemzően döntési diagramok segítségével tárolják. Az ezeken dolgozó algoritmusok gyakran hatalmas állapottereket képesek hatékonyan kezelni és tárolni.

Komplex, komponensekből felépülő rendszerek esetében komoly előrelépést jelenthet az állapottér-reprezentáció során a rendszermodell struktúrájának kiaknázása a redundancia csökkentéséhez. Az utóbbi néhány évben jelentek meg az úgynevezett hierarchikus döntési diagramok, melyek a modellek hierarchikus adatszerkezeteihez illeszkedve képesek hatékonyan csökkenteni az állapottér-reprezentáció méretét.

Jelen szakdolgozat célja a hierarchikus döntési diagramok vizsgálata és implementálása az azokon értelmezett műveletekkel és állapottér felderítő algoritmusokkal együtt, illetve ezek összehasonlítása a megszokott szimbolikus modellellenőrző technikákkal.

Abstract

In safety critical software systems, the use of design-time verification, like model checking, is more and more common. As systems become more complex, the number of state variables of the systems also grows. This leads to the state space explosion problem in these "data-intensive" systems, where the state space cannot be stored with explicit methods.

To address this problem, symbolic model checking techniques were introduced. These methods store the state space with compact data structures, such as decision diagrams. Algorithms using decision diagrams can efficiently operate on and store huge state spaces.

The exploitation of the structure of the system model can lead to improvements in the storage of state spaces of complex systems consisting of several components. During the last several years, hierarchical set decision diagrams were introduced, which are naturally fitting to the hierarchical data structures of the models – reducing the size of the state space representation.

This thesis presents a theoretical overview and a new implementation of the set decision diagrams complete with operations and state space exploring algorithms. The approach is also compared to traditional symbolic methods found in the literature.

Chapter 1

Introduction

It is evident that in a critical system, whether it be hardware or software, a fault can cause serious consequences. A faulty control unit in a car or a plane can be responsible for human lives, and even if a fault is detected before causing an accident, the recall of faulty products cause huge financial losses to the manufacturing companies.

The verification of systems during design time is an effective way to discover errors in the designs of the system, to spare significant expenses and to raise the overall quality of the product in terms of dependability. *Formal verification* is a mathematically founded way to analyze the correctness of systems, and it is getting acknowledged in the critical systems industry.

An automated technique of formal verification called model checking was introduced in the early '80s [16], and has since undergone an enormous advancement. Various kinds of algorithms have been developed and implemented, and several special methods were devised for the verification of a broad range of system types.

Even today, a great challenge to the model checking is the phenomenon that is commonly referred to as *state space explosion*. It means that even a relatively small system can have a huge number of states. One of the possible reasons for the state space explosion can be the large number of state variables. Even one additional state variable can cause an exponential growth in the number of possible states of the system. *Data-intensive systems*, i.e. systems which use lots of data can easily have state spaces that cannot be stored explicitly in the physical memory of the computers.

To address this problem, *symbolic model checking* was introduced in the '90s [3], which represents the state space symbolically by efficient data structures such as *decision diagrams*. Since the first usage of decision diagrams in model checking, a number of specialized versions and extensions of decision diagrams were developed.

In the mid 2000s, Jean-Michel Couvreur and Yann Thierry-Mieg introduced a new type of decision diagram, aiming to exploit the structure of the models, called *set decision diagram* [11]. Set decision diagram is still a less prevalent, young data structure, but it could be the base of new state-of-the-art model checkers of data-intensive systems, because of its improved memory-usage.

This thesis aims *1)* to give the reader a deeper insight into set decision diagrams, and the algorithms using them, as well as *2)* to develop and implement set decision diagrams with operations and related algorithms, and *3)* to compare these approaches to traditional symbolic methods.

The layout of the thesis is aligned according to the stated goals. After laying the historical background and motivation here, Chapter 2 presents the theoretical background of the subject. Chapter 3 gives an overview on set decision diagrams. The main contributions are presented in Chapters 4 and 5. The former gives an overview on model checking using set decision diagrams (with comparisons to the most common symbolic model checking methods) and presents new developments, which forms the theoretical contributions of this work, while the latter presents the implementation of the data structure. The evaluation of the implementations are presented in Chapter 6. Finally, Chapter 7 concludes the result, and gives possibilities for future works.

Chapter 2

Background

This chapter overviews the basic concepts and methods to lay down the basis of this work. At first, Section 2.1 introduces Petri nets, which is a modeling language used in the field of formal methods. Secondly, Section 2.2 introduces Kripke structures commonly used to describe and represent state spaces of models, including Petri nets. Thirdly, Section 2.3 presents the basic concepts of model checking. Then Section 2.4 gives an overview on the topic of decision diagrams, which are the data structures used by the majority of symbolic model checkers. Finally, Section 2.5 presents multiple algorithms for exploring the state space of a model.

2.1 Petri Nets

This thesis relies on the modeling formalism called *Petri net*. Petri nets are a widely used graphical and mathematical modeling language with the main strength of modeling asynchronous, concurrent and nondeterministic systems, making them a suitable formalism to be the basis of this work.

This section gives a brief introduction on Petri nets, for more detailed descriptions, refer to [22].

Definition 1 (Petri net).
A *Petri net* is a 5-tuple $PN = (P, T, A, W, M_0)$ where:

- P is a finite set of *places*;
- T is a finite set of *transitions*;
- $A \subseteq (P \times T) \cup (T \times P)$ is the set of directed *arcs*;
- $W : A \to \mathbb{N}^+$ is an arc *weight function*;
- $M_0 : P \to \mathbb{N}$ is the initial marking, i.e., the number of *tokens* on each *place*;
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$. ▪

A Petri net consists of places, transitions and directed arcs between places and transitions. A state of the Petri net is determined by the *marking function* ($M : P \to \mathbb{N}_0$) assigning a number of *tokens* for every place.

Arcs from places to transitions are called *input arcs*, and arcs from transitions to places are called *output arcs*. The *input places* of a transition t are denoted by $\bullet t = \{p : p \in P \wedge (p,t) \in A\}$. In contrast, the *output places* of a transition t are denoted by $t\bullet = \{p : p \in P \wedge (p,t) \in A\}$.

The behavior (i.e., the state changes) of the Petri net is defined by the following *firing rules*:

- A transition t is *enabled iff* $\forall p \in \bullet t : M(p) \geq W(p,t)$, i.e., all input places of t has at least as many tokens as the weight of the input arcs of t.
- An enabled transition may *fire*. Firing is a non-deterministic behavior, because there is no ordering or precedence amongst them, and an enabled transition may not fire at all.
- When a transition t fires, it removes $W(p,t)$ tokens from all of its input places $p \in \bullet t$ then adds $W(t,p)$ tokens to its output places $p \in t\bullet$.

The following, more illustrative notations can be also used:

- $W^+(t,p) = W(t,p)$ is the total amount of tokens added to place p by the firing of transition t.
- $W^-(t,p) = W(p,t)$ is the total amount of tokens removed from place p by the firing of transition t.
- $W^*(t,p) = W^+(t,p) - W^-(t,p)$ represents the sum of removed and added number of tokens in place p by the firing of transition t.

An example of the firing mechanism in Petri nets is shown on Figures 2.1 and 2.2.

If the ordered sequence of transitions τ can be fired in a Petri net in that exact order from a current state of the Petri net, then τ is a *firing sequence*. The sequence of states reached after each step in τ (including the initial state) is called a *path* and is denoted by ρ. A marking M is *reachable* from the initial marking M_0 *iff* there is a path ρ that begins with M_0 and ends with M.

Definition 2 (Bounded Petri net).
A Petri net is *bounded iff* $\exists k \in \mathbb{N}$ such that $\forall p \in P, M \in RM : M(p) \leq k$, where RM denotes the set of the reachable markings from the initial marking. ▪

The reachable state space is finite *iff* the Petri net is bounded. Throughout this work, it is assumed that the examined Petri net models are bounded.

Petri nets are graphically as edge weighted directed bipartite graphs. Places are represented by circles, and transitions are represented as rectangles. Arcs are drawn as directed edges between places and transitions, labeled with the weight of the arc. Weights of 1 are usually not represented. Tokens of the current marking are represented with dots inside the places. Figures 2.1 and 2.2 present graphical representation of Petri nets.

2.2 Kripke Structures

The state spaces of high-level models can be described by the so-called *Kripke structures* [20]. Kripke structures are directed graphs, with labeleded nodes. Nodes represent the

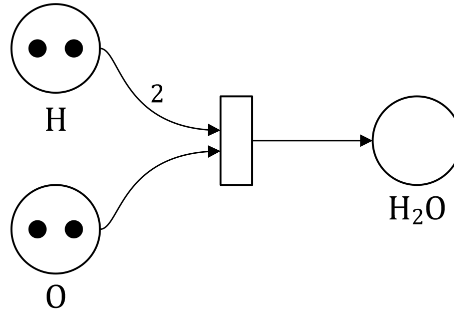

Figure 2.1. A Petri net model of the reaction of hydrogen and oxygen.

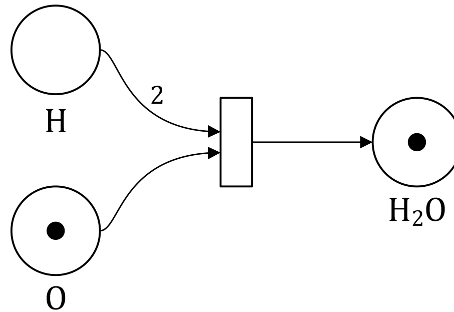

Figure 2.2. The Petri net model of the reaction of hydrogen and oxygen after firing the transition.

different states of the modeled system, while edges denote the transitions between different states. Each node is labeled with properties that hold in the corresponding state.

Definition 3 (Kripke structure).
Given a set of atomic propositions $AP = \{p, q, \ldots\}$, a (finite) *Kripke structure* is a 4-tuple $K = (S, I, R, L)$, where:

- S is the finite set of states;
- $I \in S$ is the set of initial states;
- $R \in S \times S$ is the transition relation;
- $L : S \longrightarrow 2^{AP}$ is the labeling function that maps a state to a subset of atomic propositions; ▪

In a Kripke structure a *path* (or a *trace*) ρ is directed path in the graph, corresponding to an ordered sequence of states.

2.2.1 Kripke Structures as State Space Representations

As stated in Section 2.2, Kripke structures can represent the state spaces of high level models.

When describing the state-space of a Petri net based model, a state of the Kripke structure corresponds to a marking of the Petri net. The labels on the Kripke structure are relations

interpreted over the represented marking of the labeled state. Transitions of the Kripke structure correspond to a single firing of an enabled transition in the Petri net. This method maps the state space of the Petri net to a Kripke structure, thus exploring the state space of the Petri net is equivalent to the traversal of the Kripke structure.

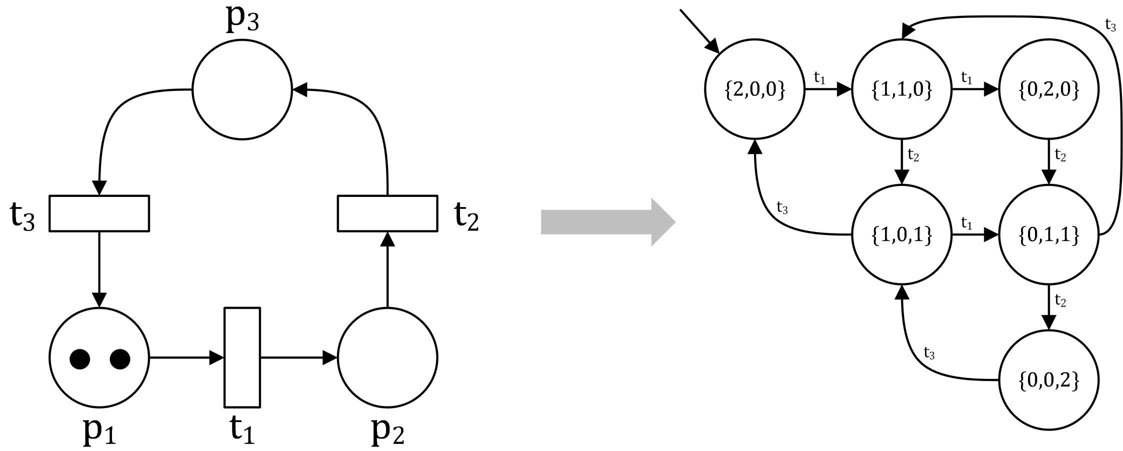

Figure 2.3. Kripke structure describing the state space of a Petri net.

As Figure 2.3 shows, the state space of a model can be larger and more complex than the structure itself – this is especially true in the case of already complex models. That means discovering and storing the full state space can be problematic.

2.3 Model Checking

Model checking [8] [7] is an automatic formal verification technique for exhaustively computing and analyzing the state space to determine if it satisfies a given requirement.

The inputs of the model checking procedure are the model of the system and formalized specifications of the expected behavior. The states or traces of the model are examined by mathematically proven algorithms and if the model violates the requirements then a counterexample are given to demonstrate the fault of the system.

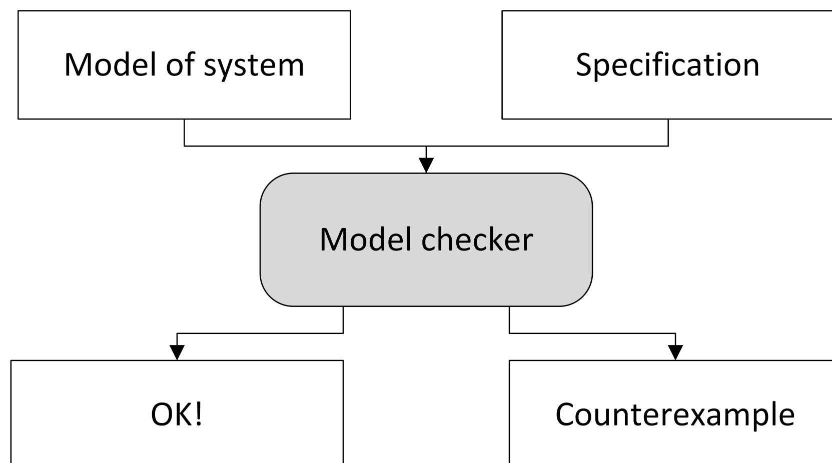

Figure 2.4. The general workflow of model checking.

2.3.1 Symbolic Model Checking

Complex systems often have a huge state space with large state vectors, so efficient encoding of states are necessary. In order to tackle the state space explosion problem, symbolic algorithms introduce special encodings of the state space. These approaches handle huge sets of states together and encode them in a compact symbolic representation.

The general idea behind symbolic algorithms is to operate on large sets of states instead of single states [1]. As stated above, these methods encode state spaces in a compact form, directly manipulating the symbolic representation. Common representations include binary functions and decision diagrams, this work focuses on the latter approach.

2.4 Decision Diagrams

Decision diagrams are more compact forms of decision trees, where the nodes with the same meaning were contracted to achieve more efficient, less redundant storage without loss of information. In this section, we introduce two of the most widely used decision diagrams: (*binary decision diagram* and *multivalued decision diagram*).

2.4.1 Binary Decision Diagrams

Binary decision diagrams (or *BDDs*) were introduced by Randal Bryant in 1986, to efficiently represent binary functions [2]. In the following we introduce BDDs in more detail as they are the most basic type of decision diagrams, encoding a set of binary vectors.

Definition 4 (Binary decision diagram).
A *binary decision diagram* is a directed acyclic graph, with a node set (V) consisting of two types of nodes: terminal and non-terminal nodes. Every nonterminal node $v \in V$ has two outgoing edges to two children nodes, we denote them as $v[0] = low(v) \in V$ and $v[1] = high(v) \in V$. Nodes are associated with levels: $level(v) \in \mathbb{Z}^+$. For every non-terminal node, $level(low(v)) < level(v)$ and $level(high(v)) < level(v)$ must hold. There are also exactly two terminal nodes, $\mathbf{0} \in V$ and $\mathbf{1} \in V$, called *terminal zero* and *terminal one* respectively. The terminal nodes also have fixed level numbers: $level(\mathbf{0}) = level(\mathbf{1}) = 0$. The terminal nodes encode binary values: $value(\mathbf{0}) = 0$, $value(\mathbf{1}) = 1$. Every BDD has a *root node* which is on the highest level (top level). ▪

The graphical representation of BDDs consists of circles for the non-terminal nodes, squares for the terminal nodes and arrows for the directed edges. The squares are labeled with the value of the corresponding terminal node. There are two type of arrows: dotted arrows for the $low(v)$ edges and solid arrows for the $high(v)$ edges.

The semantics of BDDs come from the level numbers: each level corresponds to a variable. The value of the encoded function can be computed by traversing the graph starting from the root, and following the edges according to the value of the variable of the current level.

In Figure 2.5 presents a BDD encoding a binary functions that is true for the following Boolean tuples: *(0, 0, 0)*, *(0, 1, 0)*, *(0, 1, 1)*, *(1, 0, 1)*, *(1, 1, 1)*.

2.4.2 Multivalued Decision Diagrams

Multivalued decision diagrams (or *MDDs*) are extensions of binary decision diagrams [23] [19], as seen in Definition 5.

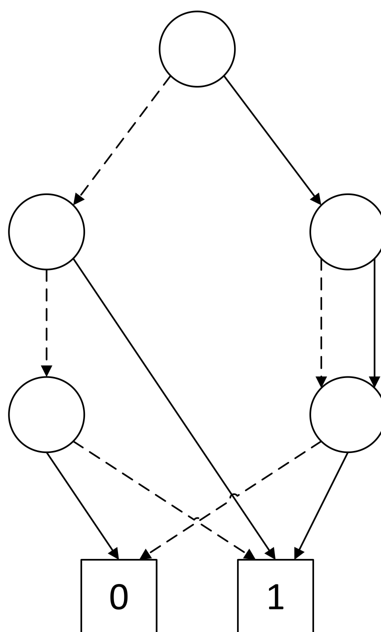

Figure 2.5. Graphical representation of a BDD.

Definition 5 (Multivalued decision diagram).
A *multivalued decision diagram* is a directed acyclic graph, with a node set (V) consisting of two types of nodes: terminal and non-terminal nodes. Nodes are associated with levels: $level(v) \in \mathbb{Z}^+$. An MDD encodes an integer function $f(x_1, ..., x_n) \longrightarrow \{0, 1\}$, and the ith level in an MDD corresponds to a variable x_i of the function with a finite integer domain $D_i = \{0, 1, 2, ..., |D_i| - 1\}$. Every nonterminal node $v \in V$ has $|D_{level(v)}|$ outgoing edges to children nodes, we denote them as $v[0], v[1], ..., v[|D_{level(v)}| - 1]$. For every non-terminal node, $level(v[i]) < level(v)$ must hold. There are also exactly two terminal nodes, $\mathbf{0} \in V$ and $\mathbf{1} \in V$, called *terminal zero* and *terminal one* respectively. The terminal nodes also have fixed level numbers: $level(\mathbf{0}) = level(\mathbf{1}) = 0$. The terminal nodes encode binary values: $value(\mathbf{0}) = 0$, $value(\mathbf{1}) = 1$. Every MDD has a *root node* which is on the highest level (top level). ▪

When there are no isomorphic sub-diagram in an MDD, we call it a *canonical MDD*. Formally, if $v = w \in V, level(v) = level(w)$ and $\forall i \in D_{level(v)} : v[i] = w[i]$ in a canonical MDD, then $v = w$.

If a canonical MDD has no edges skipping levels, we call it a *quasi-reduced MDD* [24]. Formally, this means that $\forall i \in D_{level(v)} : level(v[i]) = level(v) - 1$ holds for every non-terminal node.

If a canonical MDD has no redundant nodes, we call it a *fully-reduced MDD* [24]. Formally, this means that $\forall i, j \in D_{level(v)} : v[i] \neq v[j]$ holds for every non-terminal node.

When representing the MDD graphically, we still use circles and squares as in BDDs, but the edges in MDDs are always solid lines with an i integer label, which corresponds to the edge leading to $v[i]$. If an edge is not shown on a figure, then it either points to the terminal zero or to a *zero node*, which is a node where every outgoing *path* leads to the terminal zero.

Figure 2.6 presents an MDD encoding a function which is true for the following *(x_1, x_2, x_3)* values: *(0, 0, 1)*, *(0, 2, 2)*, *(1, 0, 2)*, *(1, 1, 2)*.

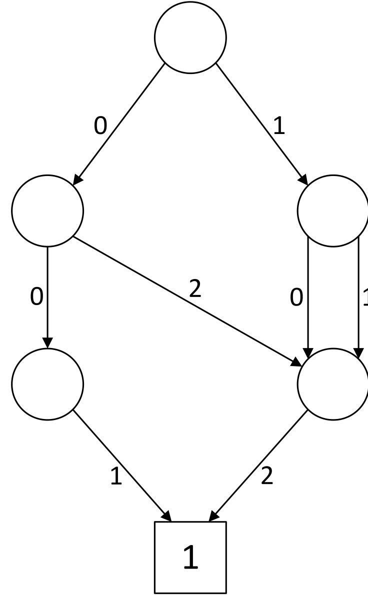

Figure 2.6. Graphical representation of an MDD.

2.4.2.1 Operations on Multivalued Decision Diagrams

MDDs are often used to encode a set of integer vectors as the function corresponding to the MDD returns **1** for exactly the vectors included in the set. Based on this, it is possible to define set operations on MDD nodes. The result of MDD operations on two MDD nodes encodes the same set as the corresponding set operations would produce from encoded sets of the original nodes. Operations are defined strictly to nodes on the same level.

The *union* of nodes v and w is

$$v \cup w = \begin{cases} v = 1 \lor w = 1 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \cup w[i]. \end{cases}$$

The *intersection* of nodes v and w:

$$v \cap w = \begin{cases} v = 1 \land w = 1 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \cap w[i] \end{cases}$$

The *relative complement* of node w in v:

$$v \setminus w = \begin{cases} v = 1 \land w = 0 & \text{if } level(v) = level(w) = 0 \\ x & \text{otherwise, where } \forall i \in D_{level(v)} : x[i] = v[i] \setminus w[i]. \end{cases}$$

The union, the intersection and the relative complement of terminal nodes w and v is similar to Boolean logic.

Due to the recursive definition of the operations, they can be efficiently realized with recursive functions. Using a cache also improves the performance, since the same nodes can be reached along multiple paths.

2.5 Algorithms for State Space Exploration

Section 2.2.1 introduced a way to represent state spaces with Kripke structures. Kripke structures are essentially directed graphs, so graph traversal algorithms can be used to explore the state spaces.

Different problems require different algorithms. For example, when determining if a given state is reachable from an initial state, then using special, directed algorithms can be more effective [15]. The main goal of the implemented methods and data structures in this thesis is to efficiently explore the whole state space, so this section will introduce general traversal algorithms.

2.5.1 Basic Algorithms

Two common methods of traversing the state space presented here are the widely known *breadth first search* (BFS), and a similar algorithm called *chaining loop*.

BFS starts at the initial state, and explores the reachable states by taking one step with every transition, from every known state and in every iteration. That means BFS finds every reachable state with the minimum required firings needed to reach that state from the initial state. With every iteration, BFS constructs the result for the firing of every enabled transition on the currently explored state space, and then it merges the results and this original set. The pseudo-code for BFS is given on Algorithm 1.

Algorithm 1: Breadth first search.

```
  Input: a set s containing the initial state only
  Output: the encoded state space
1 while new states are found do
2 |   d ← empty set;
3 |   foreach every transition t do
4 |   |   d ← d ∪ fireTransition(s, t);
5 |   end
6 |   s ← s ∪ d;
7 end
8 return s;
```

Chaining loop fires a single transition in every iteration, merging the results immediately. This means that on average, chaining loop will find most of the reachable states sooner then the BFS, but it can only give a loose upper estimation on firing needed to reach a given state. The pseudo-code for chaining loop is given on Algorithm 2.

2.5.2 Saturation

Saturation algorithm [4, 5] is a symbolic iteration strategy to explore the state spaces of concurrent systems, and it is designed to work on MDDs.

Algorithm 2: Chaining loop algorithm.

```
  Input: a set s containing the initial state only
  Output: the encoded state space
1 while new states are found do
2 |  foreach every transition t do
3 |  |  s ← s ∪ fireTransition(s, t);
4 |  end
5 end
6 return s;
```

Saturation consists of decomposing the model to components. These components can be traversed locally, which will reduce the number of global steps in the state space exploration [12]. The iteration in the saturation algorithm is adapted to the structure of the MDD representation of the state space, instead of the BFS order of the states. When saturating a decision diagram node, new nodes created on the lower levels will themselves be saturated immediately, resulting in a recursive algorithm.

The saturation algorithm is not discussed more deeply in this thesis, as it is a very complex subject on its own. For further description on saturation, refer to [4, 5, 13].

Chapter 3

Hierarchical Set Decision Diagrams

This chapter presents the so-called *set decision diagrams*, introduced by [11], aiming to represent hierarchy in the data structure. The main idea behind SDDs is that its edges encode sets of values instead of a single value. To achieve this, the outgoing edges of a node are labeled by another decision diagram node instead of an integer. The definition in [11] builds on a special decision diagram type called data decision diagram (DDD). However, the concepts of SDDs follow more naturally if they are introduced as hierarchical extensions of MDDs.

3.1 Definition of Set Decision Diagrams

The following definition of set decision diagrams builds on Definition 5.

Definition 6 (Set decision diagram).
A *set decision diagram* is a set of integer tuples represented by a directed acyclic graph, with a node set (V) consisting of two types of nodes: terminal and non-terminal nodes. Every nonterminal node $v \in V$ has at least one outgoing edge to a child node. Nodes are associated with levels: $level(v) \in \mathbb{Z}^+$. For every non-terminal node, $level(v[i]) < level(v)$ for every children of v. There are also exactly two terminal nodes, $\mathbf{0} \in V$ and $\mathbf{1} \in V$, called *terminal zero* and *terminal one* respectively. The terminal nodes also have fixed level numbers: $level(\mathbf{0}) = level(\mathbf{1}) = 0$. The terminal nodes encode binary values: $value(\mathbf{0}) = 0$, $value(\mathbf{1}) = 1$. The edges of the SDD encode sets of integer tuples. Edges are denoted by $x \xrightarrow{a_i} y$, where $x, y \in V$, and a_i is the root node of an MDD or SDD representation of the set of integers. ▪

A path from the root node of an SDD to the terminal one encodes the Cartesian product of the sets encoded by the labels of traversed edges. For the sake of convenience, the notion of a node is often used to refer to the set it encodes, if not ambiguous.

3.2 Properties of Set Decision Diagrams

In order to efficiently use SDDs they have to be unambiguous. To achieve unambiguity, canonical forms of SDDs are used.

Definition 7 (Canonical set decision diagram).
An SDD is *canonical iff*:

- $\forall v \xrightarrow{a_i} w \Longrightarrow a_i \neq \emptyset \wedge w \neq 0$;
- $\forall v \xrightarrow{a_i} w \wedge v \xrightarrow{a_j} z \Longrightarrow a_i \cap a_j = \emptyset \wedge w \neq z$. ∎

Definition 7 can be fulfilled by applying the following reduction rules. For a visual explanation, see Figure 3.1 and 3.2:

- A canonical representation of $v \xrightarrow{a_i} w$ and $v \xrightarrow{a_j} w$ is $v \xrightarrow{a_i \cup a_j} w$.
- A canonical representation of $v \xrightarrow{a_i} w$ and $v \xrightarrow{a_j} z$, where $w \cup z \neq \emptyset$ and $w \cap z \neq \emptyset$, is $v \xrightarrow{a_i \setminus a_j} w$, $v \xrightarrow{a_j \setminus a_i} z$ and $v \xrightarrow{a_i \cap a_j} w \cup z$.

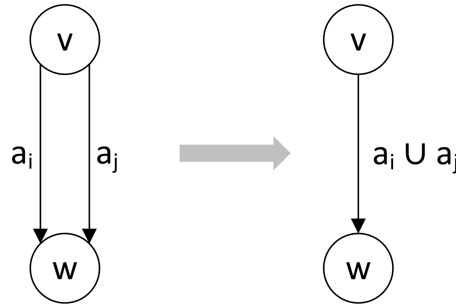

Figure 3.1. Visualization of the first SDD reduction rule.

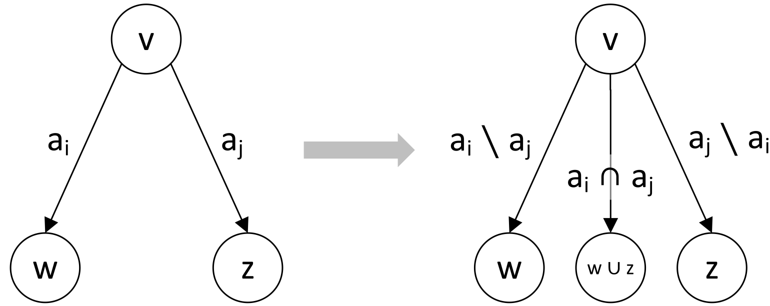

Figure 3.2. Visualization of the second SDD reduction rule.

The advantage of MDDs is that they can exploit the similarities of the encoded tuples to achieve a compact representation. In addition to this, SDDs add the capability of exploiting the inner structure of the encoded tuples, i.e., symmetries inside a tuple can be efficiently represented hierarchically. When used in symbolic model checking, this feature aligns with the compositional structure of high level models.

The graphical representation of an SDD is problematic due to the hierarchical structure. The convention used in this work is to represent labeled nodes by a dashed arrow pointing from the referencing edge to the referenced decision diagram's node. Figure 3.3 shows the graphical notations and also illustrates the source of compactness in SDDs.

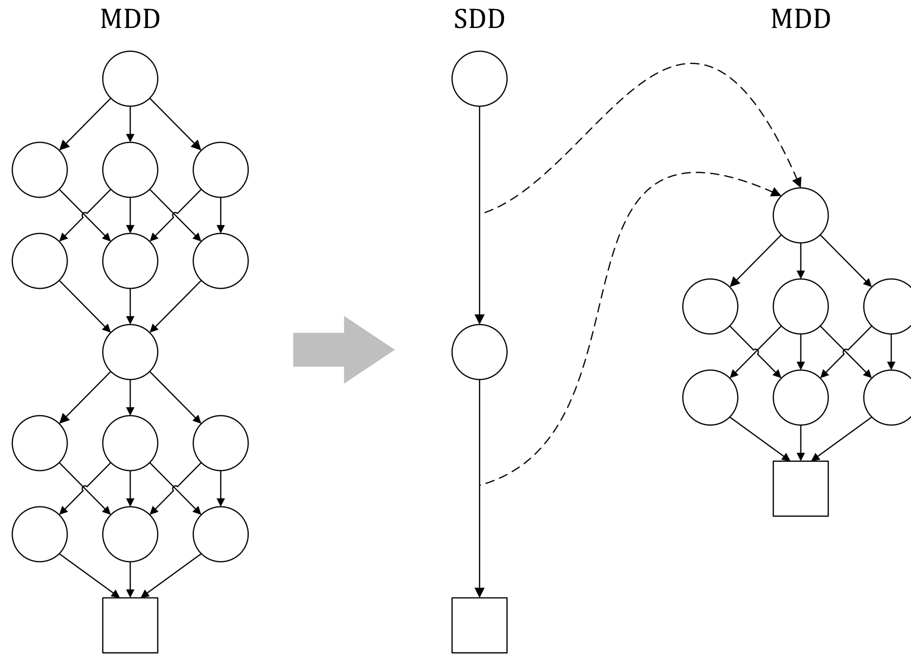

Figure 3.3. An MDD and an SDD hierarchy encoding the same set.

3.3 Operations on Set Decision Diagrams

In [11], the SDD definition builds on a special type of decision diagrams, called data decision diagrams, which are discussed further in [9]. DDDs support special types of operation, called *homomorphism.*

A homomorphism is an abstract operation that maps canonical (called well-defined by [9]) DDD nodes to canonical DDD nodes. As SDDs are generalized DDDs, [11] defines SDD homomorphisms. They can be used to define various operations, including set operations, integer arithmetic or even variable assignments.

3.3.1 Set Operations

The SDD implementation of this work is integrated into a currently developed model checker framework of the Fault-Tolerant System Research Group at Budapest University of Technology and Economics. To fit the interface of the implementation of the SDD into the framework, the usual set operations had to be defined.

The following algorithms will be doubly recursive: for every edge in the resulting diagram the label and the child node has to be computed by a recursive calls to another set operation. The recursions computing the child nodes traverse the current diagram and will be terminated on the terminal level. The recursions computing the labels to the new children traverse the hierarchy, and will eventually result in a simple MDD operation (introduced in Section 2.4.2.1). Furthermore, reduction rules must be enforced during the computation.

When defining the following algorithms, one must consider the path passing through the current decision diagram node(s). In every case we will consider how to compute the child and label nodes of the resulting edges such that the resulting node is canonical.

3.3.1.1 Intersection

The intersection of two SDDs is an SDD encoding the set of vectors encoded by both of the operands. Our first consideration is that every path that is present in the intersection has to pass trough some label in both the first and the second operand. Therefore, to compute the labels of resulting edges, we have to intersect labels of every edge of the first operand with labels of every edge of the second operand. The resulting intersection are the candidates to be labels on the edges of the result node. Furthermore if a path got through the labels, it still has to reach the terminal one in both diagrams, so the intersection of the children nodes also has to be computed for every edge candidate. If the intersection is non-empty, we add an edge to the result node labeled with the intersection of the labels leading to the intersection of their corresponding children. This construction inherently satisfies both of the reduction rules.

Algorithm 3: Intersection of two SDD nodes

```
   Input: SDD nodes v and w on the same level in two canonical SDD graph
   Output: SDD node z encoding the intersection of the inputs
 1 if v = 0 ∨ w = 0 then
 2 |  return 0;
 3 end
 4 if v = w then
 5 |  return v;
 6 end
 7 z ← new node on the same SDD level as operands;
 8 foreach outgoing edge e of v do
 9 |  foreach outgoing edge f of w do
10 |  |  c ← child(e) ∩ child(f);
11 |  |  l ← label(e) ∩ label(f);
12 |  |  if n ≠ 0 and l ≠ 0 then
13 |  |  |  create edge z -l-> c ;
14 |  |  end
15 |  end
16 end
17 if z does not have any edges then
18 |  z ← 0;
19 end
20 return z;
```

3.3.1.2 Union

The union of two SDDs is an SDD encoding the set of vectors encoded by any of the operands. Our first consideration is that every path that is present in both of the operands has to path trough some label in both the first and the second operand. But unlike in the case of intersection, the corresponding children has to be merged. Furthermore, in

this case, paths only present in one of the operands also have to be included in the result. Therefore, the parts of labels not present in the intersections has to be added as a new edge with their original child node.

While the definition of intersection implied that the reduction rules are satisfied, it is not the case here. To see this, consider the following. Assume that the label of an edge had an intersection with the label of another edge, whose child was the same. In this case the algorithm above would create two edges that lead to the same child. Therefore, the last step of the algorithm has to apply the first reduction rule (defined in Section 3.2) by merging the labels of the edges leading to the same child node. The other rule is still guaranteed by the construction.

Algorithm 4: Union of two SDD nodes

```
   Input: SDD nodes v and w on the same level in two canonical SDD graph
   Output: SDD node z encoding the union of the inputs
 1 if v = 1 or w = 1 then
 2 |   return 1;
 3 end
 4 if v = w then
 5 |   return v;
 6 end
 7 z ← new node on the same SDD level as operands;
 8 foreach outgoing edge e of v do
 9 |   foreach outgoing edge f of w do
10 |   |   c ← child(e) ∪ child(f);
11 |   |   l ← label(e) ∩ label(f);
12 |   |   if l ≠ 0 then
13 |   |   |   create edge z --l--> c;
14 |   |   |   replace e with v --label(e) \ l--> child(e);
15 |   |   |   replace f with v --label(f) \ l--> child(f);
16 |   |   end
17 |   end
18 end
19 add every remaining edges from v and w to z;
20 while ∃a_i, a_j, x : z --a_i--> x ∧ z --a_j--> x do
21 |   remove edges z --a_i--> x and z --a_j--> x;
22 |   create edge z --a_i ∪ a_j--> x;
23 end
24 return z;
```

3.3.1.3 Subtraction

The subtraction of an SDDs from another SDD encoding the set of vectors encoded only by the latter. In our implementation, we subtracted the right operand from the left operand. Our first consideration is that every path that is present in both of the operands has to path trough some label in both the first and the second operand. But unlike in the case of intersection, the corresponding children has subtracted, with subtracting the children of the right operands from the children of the left operand, thus eliminating the paths found

in both of the operands. Furthermore, in this case, paths only present in the left operand also have to be included in the result. Therefore, the parts of labels from the left operand not present in the intersections has to be added as a new edge with their original child node.

While the definition of intersection implied that the reduction rules are satisfied, it is not the case here. To see this, consider the following. Assume that the label of an edge had an intersection with the label of another edge, whose children nodes are disjunct. In this case the algorithm above would create two edges that lead to the same child. Therefore, the last step of the algorithm has to apply the first reduction rule (defined in Section 3.2) by merging the labels of the edges leading to the same child node. The other rule is still guaranteed by the construction.

Algorithm 5: Subtraction of an SDD node from another SDD node

---

Input: SDD nodes v and w on the same level in two canonical SDD graph
Output: SDD node z which is the subtraction of w from v

1 **if** $v = \mathbf{0}$ *or* $w = \mathbf{0}$ **then**
2   **return** v;
3 **end**
4 **if** $v = w$ **then**
5   **return 0**;
6 **end**
7 $z \leftarrow$ new node on the same SDD level as operands;
8 **foreach** *outgoing edge* e *of* v **do**
9   **foreach** *outgoing edge* f *of* w **do**
10         $c \leftarrow \text{child}(e) \setminus \text{child}(f)$;
11         $l \leftarrow \text{label}(e) \cap \text{label}(f)$;
12   **if** $l \neq 0$ **then**
13   create edge $z \xrightarrow{l} c$;
14   replace e with $v \xrightarrow{label(e) \setminus l} \text{child}(e)$;
15   **end**
16   **end**
17 **end**
18 add every remaining edges from v to z;
19 **while** $\exists a_i, a_j, x : z \xrightarrow{a_i} x \wedge z \xrightarrow{a_j} x$ **do**
20   remove edges $z \xrightarrow{a_i} x$ and $z \xrightarrow{a_j} x$;
21   create edge $z \xrightarrow{a_i \cup a_j} x$;
22 **end**
23 **return** z;

Chapter 4

Model Checking with Set Decision Diagrams

This chapter presents different approaches for symbolic state space representation and generation. MDD and SDD based symbolic model checking methods are demonstrated here, to illustrate the differences between the two approaches.

First, Section 4.1 shows how to use multivalued decision diagrams and set decision diagrams to encode the state space of a system, as well as some methods to represent transitions of Petri nets, or in general, vector addition systems. This section also introduces a new way to store the transitions of hierarchical models. Then Section 4.2 discusses the basic state space exploration algorithms (introduced in Section 2.5) in terms of set decision diagram representations.

4.1 Symbolic Representations

When dealing with symbolic model checking, an important question is how to encode the states and transitions of the system. In this section, two types of decision diagram based encodings are given for states, and a lightweight representation is proposed specifically for Petri net transitions.

4.1.1 Representation of the State Space

The most trivial way to represent a state space of a model is to store its state graph, explicitly enumerating the different states and transitions.

Example. Figure 2.3 illustrates a Petri net and its state space. A marking of this net can be stored as an integer vector (or tuple) of length three: the first integer referring to the token count of p_1, the second to p_2 and the third to p_3. In this case, the initial state can be represented as *(2, 0, 0)*. The state space can be stored as a set of the vectors encoding the possible states. This Petri net has six distinct states, so this approach gives the set *(0, 0, 2)*, *(0, 1, 1)*, *(0, 2, 0)*, *(1, 0, 1)*, *(1, 1, 0)*, *(2, 0, 0)* as the state space.

The example above clearly shows that the explicit storage is very straight-forward, but also very naive. Usually. there are multiple states in which the token count of a place is the same (e.g., in the example above, p_1 has zero token in three different states), which causes a redundancy. The strength of the decision diagram encoding is that it exploits the redundancy in the state space to achieve a compact storage.

4.1.1.1 Encoding States with Multivalued Decision Diagrams

MDDs are commonly used to represent sets of states, e.g., in [25]. As mentioned in Section 2.4.2, an MDD encodes an integer function $f(x_1, ..., x_n) \longrightarrow \{0, 1\}$. Suppose x_i refers to the token count of the i-th place, and $f(x_1, ..., x_n)$ is 1 *iff* $[x_1, ..., x_n]$ is part of the encoded set. The levels of the MDD then correspond to places in the Petri net. MDDs have set-like operations as seen in Section 2.4.2.1, thus it is possible to directly manipulate this representation.

Example. The Petri-net on Figure 2.3 has three places, so the MDD encoding the state space will have three levels. Let the first level represent *p1*, the second *p2* and the third *p3*. The MDD representation of the state space can be seen on Figure 4.1, with paths from the root node to the terminal **1** denotes the encoded vectors.

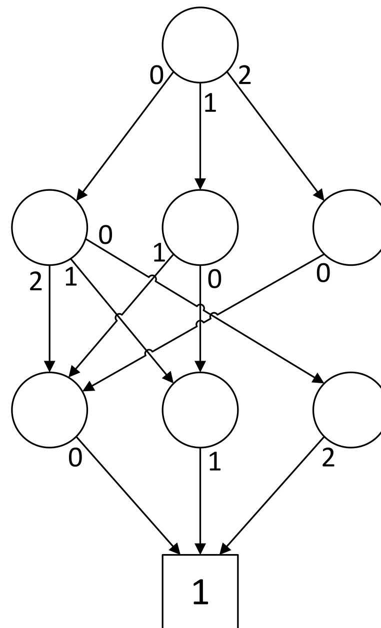

Figure 4.1. An MDD representation of the state space of the Petri net on Figure 2.3.

Figure 4.1 above illustrates the way an MDD provides a less redundant way to encode the state space. However, this method is still not redundancy-free.

4.1.1.2 Ecoding States with Set Decision Diagrams

There are models which are structurally symmetrical, and this causes redundancy in the MDD representation of their state space. Suppose a model has similar, repeated sub-models, or *components*. If these components can be arranged into a hierarchy, then we call it a *hierarchical model*.

Example. An example for hierarchical models could be the dining philosophers model. This model has multiple philosophers around a table, their meals in front of them, and a fork between every neighboring philosopher. A fork can be used by only one philosopher, and a philosopher needs both forks next to him to eat.

As Figure 4.2 shows, the problem is composed of similar, repeated components. In this case, a component consists of a philosopher and a fork. These components have similar behavior and also similar states, so the MDD has repeating patterns. Figure 4.3

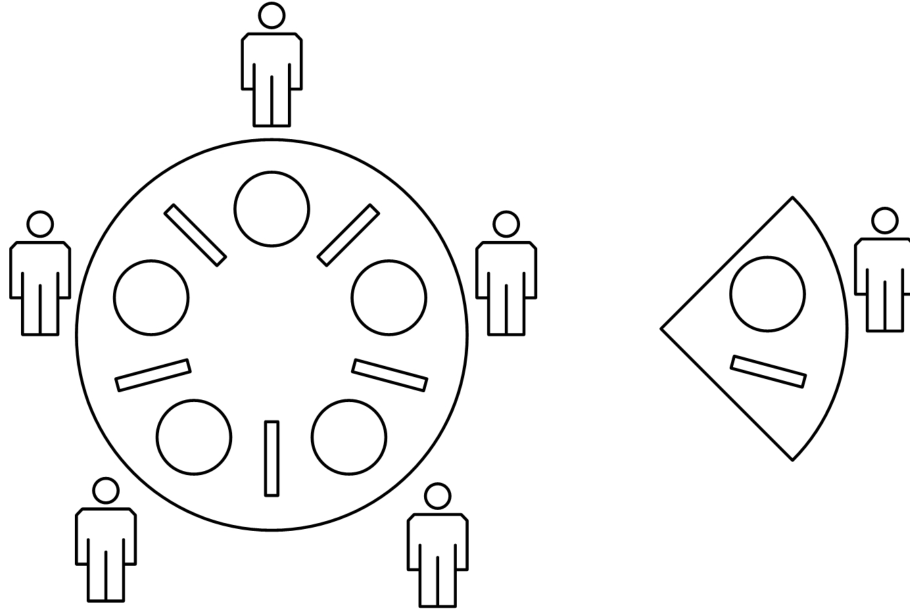

Figure 4.2. The dining philosophers problem for five philosophers.

illustrates that repeating property – the lines represent diverse sub-diagrams in the MDD (corresponding to single components), and the ellipses represent connecting, narrow parts.

To make the state space representation even more compact, set decision diagrams can be used (see Chapter 3). Using SDDs, the symmetric parts can be encoded only once, and their recurring appearance can be referenced by the use of edge labels.

Example. The aforementioned dining philosophers model can be arranged into three hierarchy levels: the topmost level, with five philosopher-fork pairs, the middle level, encoding the combination of a philosopher and a fork, and the lowest level, which encodes the states of philosophers and forks in MDDs. A route to the terminal **1** in the first hierarchy level references the second hierarchy level five times by its labels. A route in the second hierarchy level has two reference to the lowest hierarchy level (containing MDDs), with one label pointing to the MDD node encoding the states of a philosopher and another one encoding a fork.

Figure 4.4 shows a schematic figure of the resulting SDD hierarchy. Instead of storing the broad, branching parts of the state space multiple times, now they are encoded only once on the MDD level, and referenced multiple times by labels on the upper levels. Using this method, the redundancy of the state space caused by the hierarchical and repeating structure of the model is significantly reduced. For an even more detailed example on using SDD hierarchy levels, see [18].

4.1.2 Encoding of Petri Net Transitions

Symbolic algorithms generate new reachable states from the already explored part of the state space. To find these states, the algorithm has to know the transitions and their exact effect. Thus another problem arises beside the representation of the state space – the representation of transitions and their firing rules.

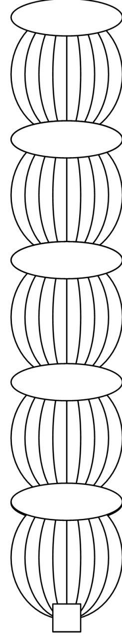

Figure 4.3. Approximate shape of an MDD encoding the state space of the dining philosophers problem.

4.1.2.1 Common Methods for Encoding Firing Rules

In case of Petri nets, the weight functions W^+ and W^- (see Section 2.1) are a sufficient representation when working with MDDs, because levels of the diagram are associated with places. A trivial representation of these weight function is a $|T| \times |P|$ input and an output *incidence matrices* N^+ and $N-$, where $N^+(t,p) = W^+(t,p)$ and $N^-(t,p) = W^-(t,p)$ [21].

Example. The Petri-net on Figure 2.3 has the following incidence matrices:

$$N^+ = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \text{ and } N^- = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Another common way to represent firing rules is the use of *Kronecker matrices* [6], which are encodings of the so-called *next-state function*. The next-state function of a model is $\mathcal{N}_t(m)$, which gives the reachable states after a single firing of transition t on state m.

Definition 8 (Kronecker matrix).
The matrix $N_t \in \{0,1\}^{|S| \times |S|}$ is a *Kronecker matrix*, where $\forall t \in T : N_t[i,j] = 1$ *iff* $j \in \mathcal{N}_t(i)$. ▪

Example. Without decomposing the Petri-net on Figure 2.3, the states of the net can be encoded as:
$M(p_1) = 2, M(p_2) = 0, M(p_3) = 0 \longrightarrow 0$
$M(p_1) = 1, M(p_2) = 1, M(p_3) = 0 \longrightarrow 1$
$M(p_1) = 0, M(p_2) = 2, M(p_3) = 0 \longrightarrow 2$
$M(p_1) = 1, M(p_2) = 0, M(p_3) = 1 \longrightarrow 3$
$M(p_1) = 0, M(p_2) = 1, M(p_3) = 1 \longrightarrow 4$
$M(p_1) = 0, M(p_2) = 0, M(p_3) = 2 \longrightarrow 5$

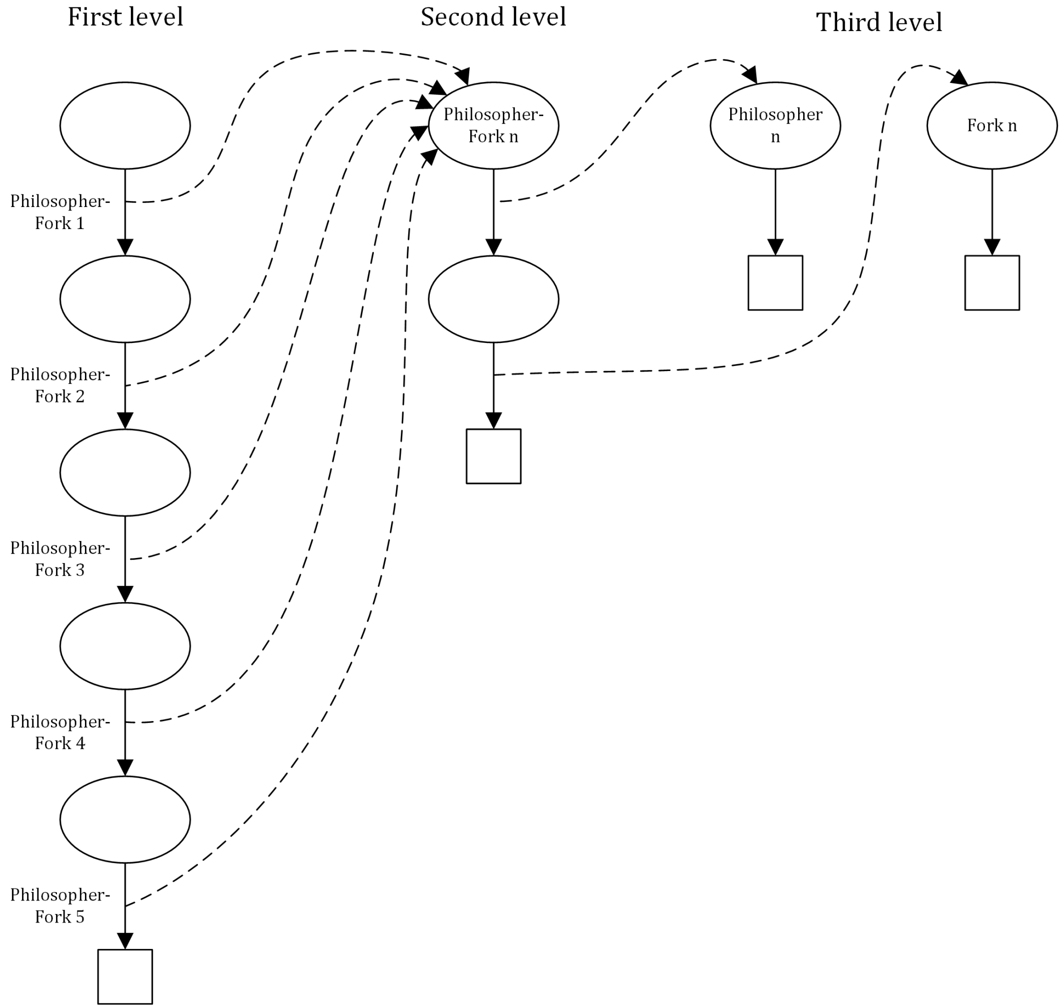

Figure 4.4. Schematich figure of a hierarchical encoding of the state space of the dining philosophers problem.

As the Kripke structure of the state space on Figure 2.3 also demonstrates, firing t_1 will take state 0 to state 1, state 1 to state 2 and state 3 to state 4, so the Kronecker matrix for t_1 is

$$N_{t_1} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The Kronecker matrices for t_2 and t_3 can similarly be constructed.

Next-state functions are often used with saturation [6, 12], and are usually decomposed for local next-state functions $\mathcal{N}_{k,t}$ according to the k cluster of the model, resulting in $N_{k,t}$ Kronecker matrices for every k level. Using Kronecker matrices in the basic exploration algorithms can be problematic, because the algorithm would have to know the reachable states beforehand to construct a matrix.

4.1.2.2 Encoding Firing Rules of Hierarchical Models

When representing the same state space with an SDD, level numbers do not correspond to places anymore, because the same diagram can encode the states of different components. In other words, the meaning of a level is now context dependent – making the encoding of firing rules with incidence matrices impossible.

To solve that problem, a method for representing the transitions hierarchically had to be developed. Because the hierarchy of a model is basically a tree structure, the transitions can be encoded in trees with the exact same structure as the model. In this construct, the missing context is reintroduced by processing the proper subtree of the transition encoding.

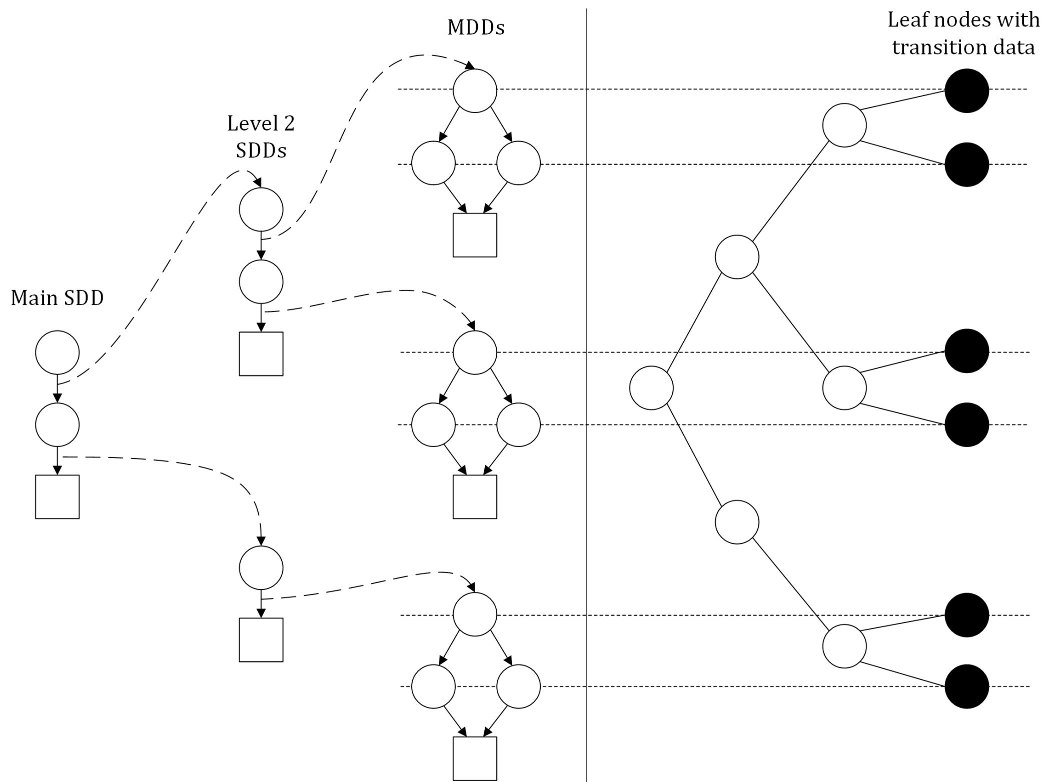

Figure 4.5. A hierarchical structure of decision diagrams, and the tree structure of the transitions.

Figure 4.5 depicts a decision diagram hierarchy along with a tree describing a transition. Leaf nodes correspond to levels of MDDs and they encode the effects of the transition in two integer values: the number of tokens removed and added to the corresponding place. Every non-leaf node corresponds to a decision diagram, with the ith child node giving context to the labels of the ith level of the SDD. This relation is formalized in the following definition:

Definition 9 (Hierarchical transition tree).
A *transition tree* describing a transition in a hierarchical Petri net is a directed tree, with a node set V consisting of internal nodes and leafs (L), an edge set E, and a function $T : L \longrightarrow (\mathbb{Z}, \mathbb{Z})$ such that $T(w) : (W(p,t), W(t,p))$ for $\forall w \in L$. Every component in the hierarchy is associated with an internal node $v \in V \setminus L$. Every place $p \in P$ in Petri net is associated with a corresponding leaf node $w \in L$. There is an edge $(x, y) \in E$ in the tree *iff* the component or place corresponding to y is the part of the subcomponent of the component corresponding to x. ▪

4.2 State Space Exploration with Set Desicison Diagrams

To traverse through the state space, an algorithm has to be defined to generate the set of states reached from an already discovered set of states after firing an enabled transition, as the state space exploring algorithms in Subsection 2.5.1 used a transition firing function. As Algorithm 6 demonstrates, the implementation of this function is simple and straightforward in the case of MDDs. This algorithm traverses down through the MDD recursively, until it finds the terminal level. If the value of an MDD edge (representing the token number of a place) minus the weight of the arc going from the place encoded by the parent node of this edge to the transition is smaller then zero, then the transition is not enabled, as this means that there are not enough tokens on the place. In this case, there is no need to call the recursion.

Algorithm 6: Simple transition firing on MDDs.

```
   Input: MDD node v, transition t
   Output: MDD node w encoding the sub-states resulted from firing transition t
 1 if level(v) = 0 then
 2 |  return v;
 3 end
 4 w ← new node on the same MDD level as the operands;
 5 p ← the place encoded by the level of v i = 0;
 6 foreach outgoing edge e of v do
 7 |  if n[i] ≠ 0 ∧ i – W⁻(t,p) >= 0 then
 8 |  |  x ← fireTransition(node[i], t);
 9 |  |  if x ≠ 0 then
10 |  |  |  w[i − W⁻(t,p) + W⁺(t,p)] ← x;
11 |  |  end
12 |  end
13 end
14 if w don't have any edges then
15 |  w ← 0;
16 end
17 return w;
```

As stated in Subsection 4.1.2, a level in a set decision diagram does not correspond to a single distinct place of the Petri net, so Algorithm 6 is not usable with SDDs. To use BFS and chaining loop with SDDs, new hierarchical firing algorithms had to be developed, which are discussed in Subsection 4.2.1. Using these algorithms, BFS and chaining loop can be applied to hierarchical models without alteration.

Saturation with SDDs is also possible, and it is empirically an order of magnitude better than the common traversal algorithms [18].

4.2.1 Firing Transitions in Hierarchical Systems

Algorithms 7 and 8 are designed to use the transition trees for firing transitions by calling doubly recursion in accordance with the structure presented in Section 4.1.2. Algorithm 7 is called on the root node of the SDD on the top hierarchy level, and when it travels down to the lowest SDD hierarchy levels, then eventually it calls the MDD variant of the algorithm, when the labels are referring to MDDs. This MDD variant algorithm (Algorithm 8) will determine if a transition is enabled, and if it is, then calculates the new token count on the place of the Petri net corresponding the MDD level within the current context. The other recursion traverses downwards in the levels of the SDD to calculate the child nodes, and terminating on the terminal level.

Algorithm 7: Fire transition algorithm for SDD nodes.

Input: SDD node v, tree node t from the transition tree of a transition
Output: SDD node w encoding the sub-states resulted from firing the transition corresponding to the tree of t on the states encoded by v

```
1  if level(v) = 0 then
2  |  return v;
3  end
4  w ← new node on the same SDD level as the operands;
5  foreach outgoing edge e of v do
6  |  n ← fireTransition(node(e), t);
7  |  l ← fireTransition(label(e), t[level(v)]);
8  |  if l ≠ 0 ∧ n ≠ 0 then
9  |  |  create edge w -l-> n;
10 |  end
11 end
12 if w don't have any edges then
13 |  return 0;
14 end
15 while ∃a_i, a_j, x : w -a_i-> x ∧ w -a_j-> x do
16 |  remove edges w -a_i-> x and w -a_j-> x;
17 |  create edge w -(a_i ∪ a_j)-> x;
18 end
19 return w;
```

Algorithm 8: Fire transition algorithm for MDD nodes.

Input: MDD node v, tree node t from the transition tree of a transition
Output: MDD node w which encodes the sub-states resulted from firing the transition corresponding to the tree of t on the states encoded by v

```
1  if level(v) = 0 then
2  |   return v;
3  end
4  w ← new node on the same MDD level as the operands;
5  i = 0;
6  foreach outgoing edge e of v do
7  |   if n[i] ≠ 0 ∧ i - t[level(v)].From >= 0 then
8  |   |   x ← fireTransition(node[i], t);
9  |   |   if x ≠ 0 then
10 |   |   |   w[i − t[level(v)].From +t[level(v)].To] ← x;
11 |   |   end
12 |   end
13 end
14 if w don't have any edges then
15 |   w ← 0;
16 end
17 return w;
```

Chapter 5

Implementation

The practical contribution of this thesis is the implementation of the SDD data structure, and its related algorithms as a new library. This chapter discusses the challenges faced during development as well as the finished data structure.

The library was implemented with the goal of being usable with the successor of the PetriDotNet [17] model checker framework developed by the Fault Tolerant Systems Research Group at the Budapest University of Technology and Economics. That required the aligning of the interface of the library to match the interfaces of the data structures already part of the program. The library was implemented with C# programming language, as the model checker of the faculty is being developed in the .NET framework.

As there are only a handful of papers available on SDDs, all of them without any words on the concrete implementation [11, 18], and to the best of our knowledge, only one public SDD package is available [10] (implemented in a fully different environment and with little documentation), the biggest challenge was that the development of the data structure (and the majority of the related algorithms) had to be started from scratch (the new algorithms are discussed deeper in Subsections 3.3.1, 4.1.2 and 4.2.1).

5.1 Overview of the Implemented Set Decision Diagram

This section overviews the classes of the SDD implementation, and their more significant methods. Figure 5.1 shows the class diagram of the SDD. This diagram excludes the classes of the MDD implementation (which is used on the lowest hierarchy level), as they were already a part of the model checker framework.

The classes and their role are the following:

- **SDDGraph** is the main class for storing an SDD graph. Every SDD on the same hierarchy level is stored in one SDDGraph class, even when they are disjoint. SDDGraphs consist of several SDDLevels. To make a new node in a graph, the method CreateNonTerminalNode should be called.
- **SDDLevel** is an abstract class, as it can be a terminal or a non-terminal level (SDDTerminalLevel and SDDVariable respectively). The operations defined in Section 3.3.1 are implemented in these classes as methods.
- **SDDTerminalLevel** is the terminal level of an SDD. It stores the two terminal nodes, zero and one.

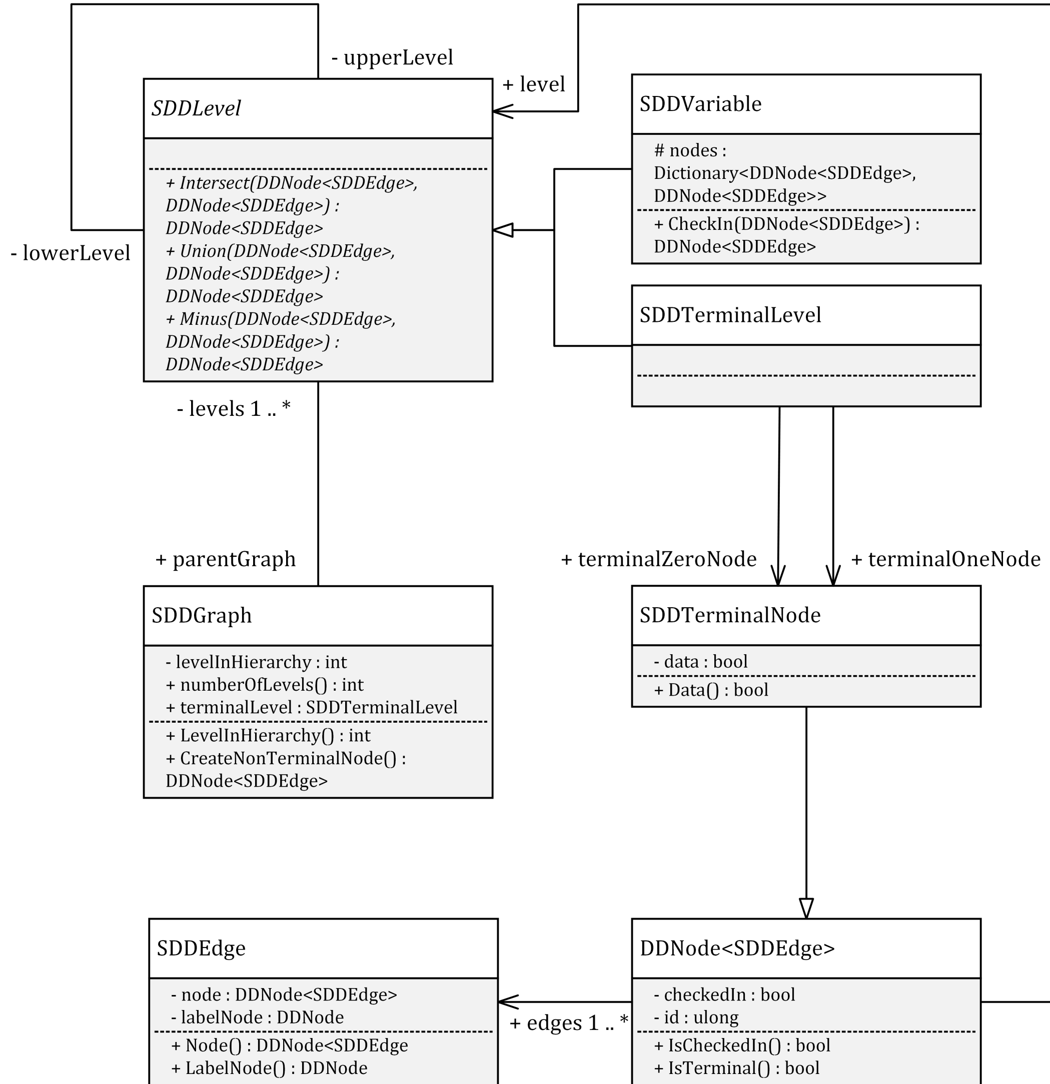

Figure 5.1. The class diagram of the implemented SDD data structure.

- **SDDVariable** is a non-terminal level in an SDD. It stores its nodes, and have a method called CheckIn, which checks if there is a node with the same outgoing edges as its input node. If it finds such a node, then it disposes the input node, and gives back the found node, otherwise it adds the input node to the stored nodes of the level. The purpose of this method is to guarantee that there is no isomorphic sub-diagrams in the decision diagram.
- **SDDEdge** is the class representing an edge in an SDD. It has a child node, and a label node. The child node is always an SDD node (implemented as a DDNode<SDDEdge>), and the label node can be an MDD node in the lowest hierarchy level, and an SDD node in every higher hierarchy level.
- **DDNode** is a universal class for decision diagram nodes. It is a generic class, with a type parameter denoting the type of the outgoing edges.
- **SDDTerminalNode** is an inheritance of the DDNode<SDDEdge>, with a single boolean value (false for **0**, and true for **1**).

5.2 Optimizations

Model checking is a complex, memory-intense task – especially when the exploration of the whole state space is needed. Thus the performance can be improved even with slight optimizations in the memory usage.

To optimize the performance of the algorithms, the implementation uses caches whenever possible: there are three caches in the SDDVariable class (one for each set operation), and a cache for every hierarchy level when firing transitions. The caches are usable, because the union, intersection, subtraction and the transition firing are static operations, which means that for the result of an operation for two nodes (or a node and a transition) can be stored to later use.

As Section 5.1 mentioned, every SDD on the same hierarchy level is stored in the same SDDGraph. This is another optimization feature, which can improve the running time of the set operations by a significant amount, as seen in [12]. The stored SDDs will not necessarily be disjoint, because the CheckIn method forbids two nodes with the same meaning. As the root node unequivocally identifies an SDD, this does not cause any malfunctions.

Chapter 6

Evaluation of the Results

This chapter presents the performance results of the implemented data structures and algorithms, and compares the different solutions. The measurements consist of the two basic algorithms presented in Subsection 2.5.1, both with using MDDs and SDDs as the representation technique of the state space.

The measurements were performed on the following system: Intel i5-3210-M CPU @ 2.50 GHz, 6 GB DDR3-1333 MHz RAM, Microsoft Windows 10 operating system with .NET platform 4.6. The memory usage of the running applications were limited to a maximum of 4 GB and enforced a time limit of 10 minutes. If a test-case violated these limitations, it was terminated and displayed the reason of termination in the results.

6.1 Results

This section gives the actual results of the implementation. The notations used in the tables are the following:

- The *Model* column contains the name of the model we ran test-case on. The description of these models can be found in appendix A
- The *|S|* column shows the size of the state-space of the model, if it is unknown in the literature, it contains the "?" symbol.
- The *RT* column shows the total runtime of the process.
- The *PMU* column shows the peak memory usage of the process.
- *> 4 GB* means that the process terminated due to too much memory consumption.
- *> 10 m* means that the process terminated due to reaching the time limit.
- if in a cell the symbol "–" showed it means that the cell cannot contain valuable data due to termination because of other reasons or that the data is not representative.

Model	|S|	SDD				MDD			
		BFS		Chaining Loop		BFS		Chaining Loop	
		RT	PMU	RT	PMU	RT	PMU	RT	PMU
Dekker 10	6 144	> 10 m	–	> 10 m	–	1.86 s	62 MB	1.88 s	63 MB
Dekker 20	278 528	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Dekker 50	?	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Peterson 2	20 754	> 10 m	–	> 10 m	–	4.24 s	124 MB	4.28 s	123 MB
Peterson 3	3.408×10^6	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Peterson 4	6.299×10^8	>10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
FMS 5	2.895×10^6	39.95 s	13 MB	9.93 s	1 MB	0.9 s	23 MB	0.79 s	23 MB
FMS 10	2.501×10^9	64.57 s	15 MB	65.42 s	11 MB	5.12 s	114 MB	0.51 s	16 MB
FMS 20	6.029×10^{12}	> 10 m	–	> 10 m	–	50.55 s	720 MB	50.04 s	699 MB
FMS 50	4.24×10^{17}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
DPhil 10	59 049	20.03 s	2 MB	8.04 s	1 MB	0.24 s	13 MB	0.07 s	5 MB
DPhil 20	3.487×10^9	> 10 m	–	194.72 s	5 MB	2.8 s	94 MB	2.68 s	94 MB
DPhil 50	7.179×10^{23}	> 10 m	–	> 10 m	–	61.54 s	1570 MB	61.95 s	1585 MB
DPhil 100	5.146×10^{47}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
Kanban 5	1.006×10^9	7.07 s	3 MB	3.49 s	1 MB	0.5 s	15 MB	0.51 s	15 MB
Kanban 10	1.006×10^9	52.07 s	14 MB	29.03 s	7 MB	3.81 s	72 MB	0.8 s	19 MB
Kanban 20	8.054×10^{11}	536.98 s	92 MB	243.13 s	50 MB	48.43 s	520 MB	47.6 s	522 MB
Kanban 50	1.043×10^{16}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB
TokenRing 5	166	69.8 s	7 MB	58.42 s	2 MB	0.18 s	7 MB	0.16 s	7 MB
TokenRing 10	58 905	> 10 m	–	> 10 m	–	14.44 s	324 MB	14.29 s	324 MB
TokenRing 20	2.477×10^{10}	> 10 m	–	> 10 m	–	–	> 4 GB	–	> 4 GB

Table 6.1. Comparison of the basic algorithms using MDDs and SDDs.

6.1.1 Evaluation of the Measures

Table 6.1 contains the measurement datas for the two different decision diagrams combined with the two different state space traversal algorithms. An interesting result is that the automated termination of the program is always caused by the runtime in case of SDDs, and by the memory usage in case of MDDs.

Unfortunately, SDDs were terminated more than MDDs, but it is still clear from looking at the peak memory usages that it is indeed a way more compact data structure than the MDD, as the memory usage in the case of using SDDs are an order of magnitude better.

The measurements reveal that the state space exploration with SDDs are more time consuming: on average, it is worse by an order of magnitude, but there are cases (like the philosophers model), where there is a two decimal difference. This drawback is probably caused by the doubly recursive property of the set operations.

In [18] it was demonstrated that the SDDs (and as seen in [14], most data structures) are far more efficient when using saturation, as it can even surpass the saturation of flat models. Implementing saturation for hierarchical structures are beyond the scope of this thesis, but it can be an interesting path for future works.

Judged from the measurements, the chaining loop proved to be the more efficient algorithm: whereas at models like TokenRing, Dekker or Peterson, there are little to no differences between the breadth first search and the chaining loop, but at DPhil, FMS and Kanban, the chaining loop have beaten the BFS with a decimal in the most cases.

Chapter 7

Conclusion and Future Work

This chapter summarizes the results and contributions of this work, and present some of my future plans of further developing the algorithms.

7.1 Contributions

The theoretical contributions of this thesis to the field of symbolic model checking are the development of the usual set operation algorithms for the set decision diagrams, detailed in Section 3.3, and the development of a method for encoding the firing rules of a hierarchical system, detailed in Subsection 4.1.2.2.

The practical contributions of this work are the implementation of set decision diagrams with its operations for the formal verification of hierarchical systems, and the implementation of two state space exploring algorithms with SDDs as well as MDDs. The finished work is integrated into the model checker framework developed at the Fault Tolerant System Research Group of the Budapest University of Technology and Economics.

7.2 Conclusion

Set decision diagram is still an uncommon, young data structure – as far as I know, this thesis is among the first few works about SDDs.

The main problem of model checking is the size of the state spaces of complex models, which is often caused by the number of state variables, thus the lightweight memory consumption of the set decision diagrams can be a big improvement to the model checkers of the future. My implementation showed that while SDDs greatly exploit the structure of models to reduce redundancy, the running time of the operations on SDDs will suffer from their hierarchical structure. To improve this performance, future works will be necessary.

7.3 Future Work

The further development and optimization of set decision diagrams is a promising future work, as the memory usage proved to be cutting-edge, while there are methods to improve the time consumption of the algorithms. Implementing saturation to hierarchical structures is a part of our future plans with SDDs, because it can greatly improve their speed as seen in [18]. Designing and implementing homomorphisms also seems to be a

viable solution to increase the effectiveness of SDDs, because these are the original operations defined on the structure. One of the most promising plans for future research is the parallelization of the processing of SDDs, exploiting the doubly recursive nature of their algorithms.

Acknowledgements

I would like to thank IncQueryLabs Ltd. for their support during my summer internship.

I wish to express my sincere gratitude to my advisor, Vince Molnár, for his support and enthusiasm during my year as his student, and for helping me to become a better engineer.

I would like to thank András Vörös, for the amount of help he gave me, and also for introducing me to the Fault Tolerant System Research Group of the Department of Measurement and Information Systems, where I spent my last year during my BSc course – which was the most profitable and enjoyable in the terms of my professional advancement.

List of Figures

2.1 A Petri net model of the reaction of hydrogen and oxygen. 11

2.2 The Petri net model of the reaction of hydrogen and oxygen after firing the transition. 11

2.3 Kripke structure describing the state space of a Petri net. 12

2.4 The general workflow of model checking. 12

2.5 Graphical representation of a BDD. 14

2.6 Graphical representation of an MDD. 15

3.1 Visualization of the first SDD reduction rule. 19

3.2 Visualization of the second SDD reduction rule. 19

3.3 An MDD and an SDD hierarchy encoding the same set. 20

4.1 An MDD representation of the state space of the Petri net on Figure 2.3. . 25

4.2 The dining philosophers problem for five philosophers. 26

4.3 Approximate shape of an MDD encoding the state space of the dining philosophers problem. 27

4.4 Schematich figure of a hierarchical encoding of the state space of the dining philosophers problem. 28

4.5 A hierarchical structure of decision diagrams, and the tree structure of the transitions. 29

5.1 The class diagram of the implemented SDD data structure. 34

List of Tables

6.1 Comparison of the basic algorithms using MDDs and SDDs. 37

List of Algorithms

1 Breadth first search. 16
2 Chaining loop algorithm. 17

3 Intersection of two SDD nodes 21
4 Union of two SDD nodes 22
5 Subtraction of an SDD node from another SDD node 23

6 Simple transition firing on MDDs. 30
7 Fire transition algorithm for SDD nodes. 31
8 Fire transition algorithm for MDD nodes. 32

Bibliography

[1] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[2] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992.

[4] Gianfranco Ciardo, Gerald Lüttgen, and Radu Siminiceanu. Saturation: An efficient iteration strategy for symbolic state—space generation. In Tiziana Margaria and Wang Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2001.

[5] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. Saturation unbound. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2619 of *Lecture Notes in Computer Science*, pages 379–393. Springer Berlin Heidelberg, 2003.

[6] Gianfranco Ciardo, Robert Marmorstein, and Radu Siminiceanu. The saturation algorithm for symbolic state-space exploration. *International Journal on Software Tools for Technology Transfer*, 8(1):4–25, 2006.

[7] Edmund M. Clarke. The birth of model checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26. Springer, 2008.

[8] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[9] Jean-Michel Couvreur, Emmanuelle Encrenaz, Emmanuel Paviot-Adet, Denis Poitrenaud, and Pierre-André Wacrenier. Data decision diagrams for petri net analysis. In Javier Esparza and Charles Lakos, editors, *ICATPN*, volume 2360 of *Lecture Notes in Computer Science*, pages 101–120. Springer, 2002.

[10] Jean-Michel Couvreur, Denis Poitrenaud, and Yann Thierry-Mieg. The libDDD library. `http://ddd.lip6.fr/libddd.php`.

[11] Jean-Michel Couvreur and Yann Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In Farn Wang, editor, *FORTE*, volume 3731 of *Lecture Notes in Computer Science*, pages 443–457. Springer, 2005.

[12] Dániel Darvas. Szaturáció alapú automatikus modellellenőrző fejlesztése aszinkron rendszerekhez [in Hungarian], 2010.

[13] Dániel Darvas, András Vörös, and Tamás Bartha. Efficient saturation-based bounded model checking of asynchronous systems. In Ákos Kiss, editor, *Proceedings of the 13th Symposium on Programming Languages and Software Tools, SPLST'13*, pages 259–273, Szeged, Hungary, 2013. University of Szeged.

[14] Dániel Darvas, András Vörös, and Tamás Bartha. Improving saturation-based bounded model checking. *Acta Cybernetica*, 21, 2014. Accepted, in press.

[15] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Partial-order reduction and trail improvement in directed model checking. *STTT*, 6(4):277–301, 2004.

[16] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In Jaco de Bakker and Jan van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin Heidelberg, 1980.

[17] Fault Tolerant System Research Group, Budapest University of Technology and Economics. The PetriDotNet webpage. `https://inf.mit.bme.hu/research/tools/petridotnet`.

[18] Alexandre Hamez, Yann Thierry-Mieg, and Fabrice Kordon. Hierarchical set decision diagrams and automatic saturation. In KeesM. van Hee and Rüdiger Valk, editors, *Applications and Theory of Petri Nets*, volume 5062 of *Lecture Notes in Computer Science*, pages 211–230. Springer Berlin Heidelberg, 2008.

[19] T. Kam, T. Villa, R.K. Brayton, and Alberto Sangiovanni-Vincentelli. Multi-valued decision diagrams: theory and applications. *Multiple-Valued Logic*, 4(1–2):9–62, 1998.

[20] Saul A. Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16(1963):83–94, 1963.

[21] ZhiWu Li and MengChu Zhou. Petri nets. In *Deadlock Resolution in Automated Manufacturing Systems*, Advances in Industrial Control, pages 17–43. Springer London, 2009.

[22] Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[23] A. Srinivasan, T. Ham, S. Malik, and R.K. Brayton. Algorithms for discrete function manipulation. In *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 92–95, Nov 1990.

[24] Tom van Dijk and Jaco van de Pol. Sylvan: Multi-core decision diagrams. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035 of *Lecture Notes in Computer Science*, pages 677–691. Springer Berlin Heidelberg, 2015.

[25] András Vörös, Dániel Darvas, Attila Jámbor, and Tamás Bartha. Advanced saturation-based model checking of well-formed coloured Petri nets. *Periodica Polytechnica, Electrical Engineering and Computer Science*, 58(1):3–13, 2014.

Appendices

Appendix A

Description of Tested Models

This appendix gives a short overview on the tested benchmark models used in 6. All of these tested models were used at the Model Checking Conference, 2013.

Dekker N

This model is a Petri net variant of Dekker's algorithm for the mutual exclusion problem, generalized for N processes. It is parametrized by the number of processes it realizes the algorithm on.

FMS N

This Petri net is extracted a benchmark used for SMART. It models a flexible manufacturing system with three conveyor belt. The model ha three processes, and it is parametrized with the starting token-count on each processes (conveyor belts), so the size of the model is not depends on the value of the parameter.

Peterson N

This is a model of the Peterson's algorithm for the mutual exclusion problem, in its generalized version for N processes. This algorithm is based on shared memory communication and uses a loop with N-1 iterations, each iteration is in charge of stopping one of the competing processes.

Kanban N

This Petri net is extracted a benchmark used for SMART. It models a Kanban scheduling system. Ut is parametrized with the token-counts of the places in the initial marking, so the size of the model is not depends on the value of the parameter.

DPhil N

This is the Petri net model of the famous dining philosophers problem, which is an example often used to illustrate an inappropriate use of shared resources generating deadlocks. N philosophers share a table each with N plate, and with a fork between every two neighboring plate. They are thinking and, when they need to eat, they go to the table, grab one fork from one side of their plate, then the second from the other side, then eat, and then go back thinking. If a philosopher grabbed a fork, he cannot go back thinking, until he has eaten.

TokenRing N

A complex model of a system where a set of N machines are placed in a ring. The objective of the protocol is to reach a stable state for the system, where given criteria are fulfilled.