UNIVERSITY OF ŽILINA
FACULTY OF MANAGEMENT SCIENCE AND INFORMATICS

RELIABILITY ANALYSIS OF COMPLEX SYSTEMS USING DECISION DIAGRAMS

Dissertation thesis

Registration number: 28360020243008

Study program:	Applied Informatics
Field of study:	Informatics
Workplace:	Department of Informatics Faculty of Management Science and Informatics, University of Žilina
Supervisor:	doc. Ing. Miroslav Kvaššay, PhD.

Žilina, 2024 **Ing. Michal Mrena**

Acknowledgment

First, I would like to thank my supervisor Associate Professor Miroslav Kvaššay, PhD. for his academic and personal guidance during my studies. I would also like to thank all my colleagues for creating a welcoming working environment and valuable discussions over countless lunches and hot beverages.

Furthermore, I would like to thank my parents, without whom my studies would not have been possible. To my brother for sophisticated vocational conversations. And finally, to all my family and friends for their support and encouragement.

ABSTRAKT

Analýza spoľahlivosti systémov je zložitý proces zahŕňajúci mnoho úloh. Mnohé systémy, s ktorými sa v praxi stretávame, označujeme ako komplexné systémy. Okrem bežných úloh a problémov analýzy spoľahlivosti sa musíme pri analýze komplexných systémov vysporiadať s veľkým rozsahom takýchto systémov, rôznorodosťou komponentov a výberom efektívnych algoritmov. Kľúčovým nástrojom analýzy je štruktúrna funkcia, ktorá popisuje topológiu systému. Efektívna reprezentácia štruktúrnej funkcie komplexných systémov je preto dôležitou súčasťou analýzy. V práci sa zameriavame na reprezentáciu štruktúrnej funkcie pomocou rozhodovacích diagramov, ktoré dokážu reprezentovať aj rozsiahle funkcie. Efektívna tvorba a spracovanie diagramov sú preto hlavnými témami tejto práce. Skúmanie vlastností štruktúrnej funkcie nám umožňuje skúmať vlastnosti systému, ktorý funkcia popisuje. Práca sa preto venuje analýze a efektívnej implementácii existujúcich algoritmov. Ďalej práca predstavuje niekoľko vylepšení existujúcich algoritmov, ktoré majú za cieľ zrýchlenie algoritmov alebo uľahčenie ich použitia. Hlavnými prínosmi práce sú predstavenie nového univerzálneho algoritmu na výpočet logických derivácií, úprava existujúcich algoritmov na pravdepodobnostnú analýzu, ktorá umožňuje použitie týchto algoritmov s časovo závislými pravdepodobnosťami stavov komponentov s použitím symbolických výpočtov. Posedným dôležitým prínosom je implementácia softvérového nástroja na analýzu spoľahlivosti s použitím rozhodovacích diagramov, ktorý implementuje všetky navrhnuté a upravené algoritmy.

Kľúčové slová: analýza spoľahlivosti; binárny rozhodovací diagram; časovo závislá analýza spoľahlivosti; pravdepodobnostná analýza spoľahlivosti; softvérové spracovanie rozhodovacích diagramov; štruktúrna funkcia; viachodnotový rozhodovací diagram

ABSTRACT

System reliability analysis is a complicated process involving various tasks. Many of the systems we encounter in practice are referred to as complex systems. In addition to the usual reliability analysis tasks and problems, when analyzing complex systems, we have to deal with the large scale of such systems, the variety of their components, and the selection of efficient algorithms. A key analysis tool is the structure function, which describes the topology of the system. An efficient representation of the structure function of complex systems is, therefore, an important part of the analysis. The thesis focuses on the representation of the structure function using decision diagrams, which can also represent large-scale functions. Efficient diagram creation and processing are therefore the main topics of this thesis. Exploring the properties of the structure function allows for the investigation of the properties of the system that the function describes. Therefore, the thesis deals with the analysis and efficient implementation of existing algorithms. Furthermore, the thesis presents several improvements to the existing algorithms that aim to make the algorithms faster or easier to use. The main contributions of the thesis are the introduction of a new universal algorithm for the computation of logical derivatives, and the modification of existing algorithms for probabilistic analysis, which allows the use of these algorithms with time-dependent probabilities of the states of the components using symbolic computations. Finally, an important contribution is the implementation of a software library for reliability analysis using decision diagrams. The open-source library implements all the proposed and modified algorithms.

Keywords: Binary Decision Diagram; Multi-valued Decision Diagram; probabilistic reliability analysis; reliability analysis; software processing of decision diagrams; structure function; time-dependent reliability analysis

Contents

List of Images

List of Tables

List of Algorithms

Nomenclature and Acronyms

Acronyms

ADD	Algebraic Decision Diagram
AST	Abstract Syntax Tree
BDD	Binary Decision Diagram
BDT	Binary Decision Tree
BI	Birnbaum's Importance
BSS	Binary-State System
CI	Criticality Importance
DNF	Disjunctive Normal Form
DPBD	Direct Partial Boolean Derivative
DPLD	Direct Partial Logic Derivative
DT	Decision Tree
EVBDD	Edge-Valued Binary Decision Diagram
FVI	Fussell-Veselys's Importance
IM	Importance Measure
IPBD	Inverse Partial Boolean Derivative
IPLD	Inverse Partial Logic Derivative
ITE	If-Then-Else
MCS	Minimal Cut Set
MPS	Minimal Path Set
MSS	Multi-State System
MDD	Multi-valued Decision Diagram
MTBDD	Multi-Terminal Binary Decision Diagram
MVL	Multiple-Valued Logic
RBD	Reliability Block Diagram
ROBDD	Reduced Ordered Binary Decision Diagram
RODD	Reduced Ordered Decision Diagram
ROMDD	Reduced Ordered Multi-valued Decision Diagram
SI	Structural Importance
SoP	Sum of Products

General Notation

$a, b, \ldots$	value from the set $\{0,1,\ldots,m-1\}$ or $\{0,1,\ldots,m_i-1\}$
$(a_i, \boldsymbol{x})$	input vector where i^{th} component has value a
$\alpha_{\mathcal{p}}$	number of state vectors corresponding to path $\mathcal{p}$
α_ϕ	total number of state vectors – the size of the domain of ϕ
$\alpha_{\phi,j}$	number of state vectors for which the structure function evaluates to j
$f(a_i, \boldsymbol{x})$	cofactor of integer function f with respect to variable x_i and value a
i_A	index of the variable associated with internal node A
k	number from the set $\{0,1,\ldots,m_{i_A}-1\}$ indexing edges of internal node A
m	size of the codomain of integer function; the number of system states
m_i	size of the domain of i^{th} variable; the number of states of i^{th} component
n	number of variables; number of system components
$\mathcal{p}$	path in MDD
$\rho_{\mathcal{p}}$	probability of path $\mathcal{p}$
$\mathcal{s}$	number of nodes in MDD
$A, B, \ldots$	internal or terminal node
$A_k, B_k, \ldots$	k^{th} son of internal node $A, B, \ldots$
$\mathcal{I}'_{\mathcal{p}}$	set of indices of variables that are not present in path $\mathcal{p}$
$\mathcal{P}_j$	set of all paths leading to the terminal node representing value j
$\mathbb{P}_{n,m}$	matrix of component state probabilities
T_j	a terminal node representing value j

Introduction

Reliability analysis of a system is a complicated process involving several steps. Considering the nature of the system and the aim of the analysis, we can describe the system either as a Binary-State System (BSS) [1], [2] or a Multi-State System (MSS) [3], [4]. The literature offers different mathematical tools for the description of such systems. The one that we focus on in this thesis is called structure function [2], [5]. The structure function is a mapping from the states of components of the system to the state of the entire system. The function alone allows us to perform a topological analysis [6] of the system allowing us to compare different system topologies. Furthermore, if component state probabilities are available, we can perform a probabilistic analysis and compute more system characteristics such as system availability [1], [2]. Also, we can compute various important measures that quantify how individual components influence the reliability of the system [7].

Depending on the type of the system, the structure function is either a Boolean function [8], a Multiple-Valued Logic (MVL) function [9], or an integer function [10]. Software processing and analysis of such functions require a suitable representation. One such representation is a decision diagram. A decision diagram is a directed acyclic graph that is designed for the efficient representation of discrete functions. Two basic types of decision diagrams exist. The first, simpler, type is the Binary Decision Diagram (BDD) [11] designed for the representation of Boolean functions. The second, more general, type is the Multi-valued Decision Diagram (MDD) [12] introduced for the representation of MVL functions and integer functions. BDDs and MDDs can be used to represent the structure functions of BSS and MSS respectively.

Systems subjected to reliability analysis exist in different topologies and configurations. Some of those systems are regarded as complex systems. The complexity may originate from different properties of the system. For example, having components of different natures or having dependent components. Moreover, systems consisting of numerous components are also regarded as complex. Such properties increase the complexity of the structure function representing the system, which, consequently, complicates the analysis of such systems. Therefore, the development of new and improvement of existing algorithms and approaches to account for increasing complexity is an actual and important problem in reliability analysis.

Decision diagrams are generally regarded as a very efficient representation of the structure function, however, the nature of complex systems and the ongoing increase in complexity pose pressure on the continuous improvement of existing techniques and the design of new approaches. Therefore, the principal goal of this thesis is *the optimization of the application of decision diagrams in the reliability analysis of complex systems*, which results in the following research topics:

- analysis of existing approaches and algorithms utilized in the representation of the structure function by decision diagram and in their subsequent analysis;
- implementation of a performant and robust software library for the creation and manipulation of decision diagrams;
- evaluation, adjustment, and improvement of existing algorithms for the creation and manipulation of decision diagrams;
- creation of new decision diagram algorithms and methods specialized for the use case of topological and probabilistic reliability analysis.

The thesis has the following structure. Chapter 1 introduces the basics of a general system reliability analysis process. It starts with the description of the principal steps of the analysis, starting with the identification of the system type description in the form of the structure function. Then it proceeds with the presentation of typical system structures with the emphasis on the properties of a complex system. Finally, the last part focuses on the presentation of various system reliability characteristics, topological analysis, probabilistic analysis (time-independent and time-dependent variants), and importance analysis.

Chapter 2 deals with discrete functions and their relation to reliability analysis. It starts with definitions of selected discrete function types – namely the Boolean function, MVL function, and integer function – that are relevant to the reliability analysis described in Chapter 1. A considerable part of the chapter that follows focuses on logical differential calculus (specifically the logic derivatives) as a powerful tool for the analysis of discrete functions followed by the description of its applications in the aforementioned reliability analysis. The last part of the chapter introduces selected representations of discrete functions with an emphasis on their efficiency – which introduces the content of the next chapter.

Chapter 3 introduces the core topic of the thesis which is the decision diagram. It starts with the theoretical description – starting with BDDs and proceeding to the most general form used in the thesis – MDD representing an integer function. The main part of the chapter deals with practical aspects of the implementation of decision diagrams and also introduces our supporting software tool TeDDy – which is one of the practical contributions of the thesis. Then it focuses on general MDD manipulating algorithms. The chapter concludes with a description of algorithms that are designed specifically for reliability analysis with the utilization of decision diagrams. Finally, the chapter also presents a new algorithm for the dynamic merging of decision diagrams and a generalized algorithm for the calculation of system state frequencies – which are one of the new results of the thesis.

Chapter 4 is fully dedicated to the evaluation of research problems introduced in Chapter 3. Each of the problems contributes either to the improvement of dynamic diagram creation or diagram evaluation. The chapter contains experimental evaluations of existing algorithms as well as novel algorithms. The first experiment deals with the analysis of the order of diagram merging and its influence on the dynamic creation process. The following experiment verifies that the generalized algorithm for diagram merging proposed in Chapter 3 is applicable in practice and simplifies the merger of diagrams using d-ary operations. The next experiment shows that the algorithm that we proposed for the calculation of system state frequencies is the simplest and fastest solution for BSS as well as MSS. Finally, the last contribution that the chapter introduces is a new universal algorithm for the calculation of any logic derivative. A description of the algorithm is followed by an experimental comparison with generic approaches – which shows that our algorithm is considerably faster. We consider this as one of the significant contributions of the thesis.

Finally, Chapter 5 addresses research problems that deal with the probabilistic reliability analysis. It first introduces two existing principal approaches to the calculation of so-called node traversing probabilities, which is an essential part of the probabilistic analysis. Then, it proceeds with the experimental comparison of the two approaches. The contribution obtained from the comparison presents use cases that are suitable for each of the two approaches – showing that both approaches are worth implementing in software tools and that their correct usage can improve the speed of probabilistic evaluation. Furthermore, the chapter deals with time-dependent component state probabilities. It discusses two possible approaches. The first, simpler, one uses the existing algorithms with little modifications and the second uses manipulation of symbolic expressions. It concludes with a comparison of the two approaches showing that a simpler approach performs significantly better. However, the contribution lies in the description and verification of the utilization of symbolic expressions and their possible advantages.

1 Reliability Analysis

A system is a general term describing an entity consisting of components. A component is a further indivisible part of the system, which contributes to the functioning of the system. Thus, the state of the components determines the state of the system. Typical reliability analysis involves tasks such as identification of the importance of individual components, identification of situations that cause degradation of system performance, or designing the system to meet certain requirements. This chapter deals with the description of the steps of the reliability analysis process.

1.1 Number of system states

The first step in the reliability analysis is to identify the number of system states. In this step, we need to consider the properties of the examined system as well as the aim of the analysis. In the following sections, we introduce two principal approaches to the description of the number of system states.

1.1.1 Binary-State Systems

The first and the simplest approach is to consider a system to be a Binary State System (BSS) [1], [2]. A BSS can be in one of two states that are functioning and failed, often denoted using numbers 1 and 0 respectively. The decision is clear for systems that are binary in their nature. An example of such a system is a logic circuit [13], [14] where the components – logic gates – can either function or not. The binary state approach is also suitable for a system in which even a slight degradation from the perfectly functioning state can cause disaster or damage. Typical examples of such systems include nuclear power plants [15] or aviation systems [16]. Naturally, we can use the BSS approach for a system that does not belong to either one of the above-mentioned types, for example, for a system where performance levels are not discrete. In such a case a principal task is to find a threshold separating working and failed states.

1.1.2 Multi-State Systems

One of the advantages of BSS is the simplicity of the model. However, the binary approach does not suit well for all the system types. Many systems can operate at several discrete performance levels. A representative example of such systems is different types of distribution networks [17] that operate using their full capacity but can also operate at multiple levels of reduced capacity. We usually describe the states as e.g., perfectly functioning, functioning, and failed and denote them using 0 for the failed state, $m-1$ for the perfectly functioning state (where m is the number of states), and natural numbers in between 0 and $m-1$ for the intermediate states. Because of the multiple states, we designate such systems as Multi-State Systems (MSS).

The number of components and system states can vary depending on the type of the system. Consequently, we recognize two types of MSS. The first type is homogeneous MSS where each component and the system itself have the same number of states. On the other hand, nonhomogeneous MSS and its components can have a different number of states. We usually encounter nonhomogeneous MSS when we examine systems that consist of components that are different in their nature. An example of such systems is healthcare systems [18], [19] that usually include humans, hardware, and software.

1.2 Structure Function

1.2.1 Structure Function Definition

After the identification of the number of system and component states, we proceed with the creation of a mathematical description of the examined system. The description must include the dependency of the state of the system on the state of its components. We call such a description a structure function. The structure function of a BSS is a mapping of the following form [1], [20]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x}) \colon \{0,1\}^n \to \{0,1\}, \tag{1.1}$$

where n is the number of components of the system, x_i models state of the i^{th} component for $i = 1,2 \dots, n$ and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ is a state vector that holds the states of all components. Later in the thesis, we will show that the definition (1.1) agrees with the definition of the Boolean function.

We can view a homogeneous MSS as a generalization of BSS. Therefore, its structure function is a similar mapping of the form [5]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x}) \colon \{0,1, \dots, m-1\}^n \to \{0,1, \dots, m-1\}, \tag{1.2}$$

where n is the number of components of the system, m is the number of system and component states, x_i models state of the i^{th} component for $i = 1,2 \dots, n$ and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ is the state vector. Finally, we consider a nonhomogeneous MSS where components and the system can have a different number of states. Its structure function is a further generalized mapping of the form [5]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x}): \{0,1,\dots,m_1 - 1\} \times \dots \times \{0,1,\dots,m_n - 1\} \\ \rightarrow \{0,1,\dots,m-1\}, \tag{1.3}$$

where n is the number of components of the system, m is the number of system states, m_i is the number of states of the ith component, x_i models state of the ith component for $i = 1,2 \dots, n$ and $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ is the state vector. Definitions (1.2) and (1.3) agree with the definition of the Multiple-Valued Logic function and integer function that we will describe in the following chapter.

Since the structure function describes a system, we can study the properties of the system by studying the properties of the structure function. One of the properties is the monotonicity of the function. If the structure function of a system is monotonic (non-decreasing), we say that the system is coherent [1], [21] i.e., there are no situations in which failure or degradation of performance of a component results in repair or improvement of the performance of the system. The opposite of a coherent system is a noncoherent system. For a structure function of a BSS to be monotonic, it must hold for each pair of state vectors of the form $(\cdot_i, \boldsymbol{x})$ that:

$$\phi(1_i, \boldsymbol{x}) \geq \phi(0_i, \boldsymbol{x}), \tag{1.4}$$

where the notation $(a_i, \boldsymbol{x})$ denotes a state vector where the value of $x_i = a$. Similarly, for a structure function of an MSS to be coherent, it must hold for each pair of state vectors of the form $(\cdot_i, \boldsymbol{x})$ that:

$$\phi(s_i, \boldsymbol{x}) \geq \phi((s-1)_i, \boldsymbol{x}), \tag{1.5}$$

where $s = 1,2, \dots, m_i - 1$.

1.2.2 Cut and Path Sets

One of the useful characteristics of a system is a set of components whose simultaneous operation or failure is essential for the state of the system. Two significant types of sets for the reliability analysis are minimal cut sets and minimal path sets. A cut set of a BSS is a set of its components whose simultaneous failure causes failure of the system given that the system was operational. For a set to be a Minimal Cut Set (MCS) [22] it must hold that removal of any component from the set would result in the set no longer being a cut set [1], [7] i.e., if we consider that all the components of the MCS are failed then a repair of any of the components causes the system to become functional. Each MCS has a corresponding state vector known as a Minimal Cut Vector (MCV) [23].

To extend the idea of the MCSs and MCVs to MSS we consider each system state individually. Both MCS and MCV can be generalized for MSS, but the generalized definition is more intuitive when we consider MCV. We say that a state vector is MCV with respect to state j of the system if an improvement of the state of any component whose state can be improved (the component is not perfectly functioning) causes the system to reach a state at least j given that the system is in a state worse than j for the state vector.

Analogous to cut sets are path sets. A path set of a BSS is a set of components whose simultaneous functioning ensures the functioning of the system. The set is a Minimal Path Set (MPS) [22] if the removal of any component from the set would cause the set to no longer be a path set [1], [7]. A state vector corresponding to an MPS is called a Minimal Path Vector (MPV). MPV can also be generalized for MSS. We say that a state vector is MPV with respect to state j of the system if degradation of performance of any component causes degradation of performance of the system to a state worse than j given that the system is in state j or better for the state vector.

1.3 Basic System Types

Real-world systems exist in different topologies and configurations. Some are simple and we encounter them either as standalone systems or as a part of other systems and some are more complicated because of their properties or size. In this section, we introduce typical examples of both kinds.

1.3.1 Series and Parallel Systems

Series and parallel systems are one of the simplest system types. Series BSS is functioning if and only if all its components are functioning. Similarly, parallel BSS is functioning if and only if at least one of its components is functioning. Fig. 1.1 shows Reliability Block Diagrams (RBD) representing a series and a parallel system consisting of three components. The system is functioning if and only if there exists a path in the diagram connecting left and right black circles and all the components on the path are functioning.

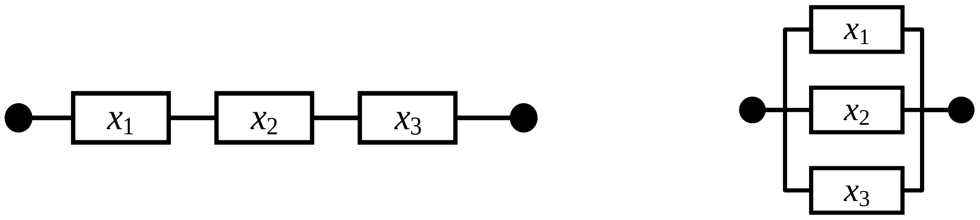


Fig. 1.1 Reliability Block Diagrams depicting series system (left) and parallel system (right)

The structure function of a series BSS has the following form [1], [2]:

$$\phi_{\text{serial}}(\boldsymbol{x}) = \bigwedge_{i=1}^{n} x_i, \tag{1.6}$$

where the $\wedge$ denotes logical conjunction (AND) and n is the number of components. Similarly, the structure function of a parallel BSS has the following form [1], [2]:

$$\phi_{\text{parallel}}(\boldsymbol{x}) = \bigvee_{i=1}^{n} x_i, \tag{1.7}$$

where the $\vee$ denotes logical disjunction (OR) and n is the number of components.

Naturally, MSS also exists in series and parallel topologies. The literature offers several functions that we can use to represent series and parallel connections. For the series topology, a sensible option is to use the min function (that returns the minimum of its arguments). To explain the rationale behind the min function let us consider the distribution network depicted in Fig. 1.2 and assume that each edge can be in one of three states offering different transportation capacities. Obviously, the throughput of the network from the source node (A) to the sink node (B) is limited by the edge with the lowest capacity – the minimum of the capacities of all edges.

A reasonable function to use for the parallel topology is the max [3], [24] function (returns the maximum of its arguments). To rationalize the choice function let us consider the distribution network depicted in Fig. 1.3 and, again, assume that each edge can be in one of three states offering different transportation capacities. Also, let us assume that the processing capacity of node B is limited – it can process at most c_{max} units, where c_{max} is the maximum of the capacities of the edges. Hence, the throughput of the network is the maximum of the capacities of the edges.

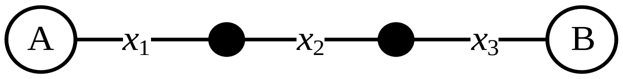


Fig. 1.2 Distribution network with series topology and unreliable edges

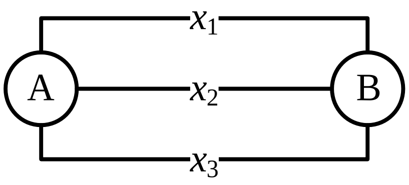


Fig. 1.3 Distribution network with parallel topology and unreliable edges

Considering an alternative where the processing capacity of node B is not limited, we may see another sensible alternative, which is to use the sum function (returns the sum of its arguments) to describe the throughput.

1.3.2 Series-parallel Systems

We usually encounter series and parallel systems as parts of a more complicated system type – a series-parallel system, which is a result of combining series and parallel topologies. Since the system is a combination of series and parallel systems, we can use the properties of those systems to describe the series-parallel system. Its structure function is therefore a composition of AND and OR operations in the case of BSS and e.g., min and max operations in the case of MSS. Let us consider the system depicted using RBD in Fig. 1.4. Assuming the system is BSS, its structure function has the following form:

$$\phi(\boldsymbol{x}) = x_1 \wedge (x_2 \vee x_3). \tag{1.8}$$

Notice that instead of a variable, the second argument of the $\wedge$ operator is the $(x_2 \vee x_3)$ expression. Using such nesting expressions, we can describe any series-parallel system.

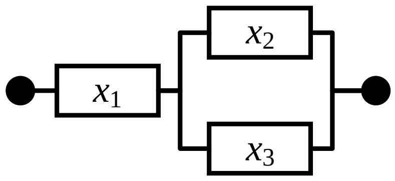


Fig. 1.4 Reliability Block Diagram depicting series-parallel system consisting of three components

RBD allows us to neatly visualize the MCSs and MPSs of a system. Let us consider the system depicted in Fig. 1.4 with two MCSs $\{1\}$ and $\{2,3\}$ with corresponding state vectors $(0,1,1)$ and $(1,0,0)$ respectively. The notation $\{2,3\}$ denotes a set containing the second and third components respectively. Notice that if any of the 0 elements in the vector would improve to 1 the system would become operational. Fig. 1.5 shows the MCSs in the RBDs using the grey color for the elements of the set. Furthermore, let us consider the state vector $(0,0,0)$. The vector corresponds to the cut set $\{1,2,3\}$. The set is not MCS because if we remove, for instance, the component 1 from the set the resulting set will still be a cut set.

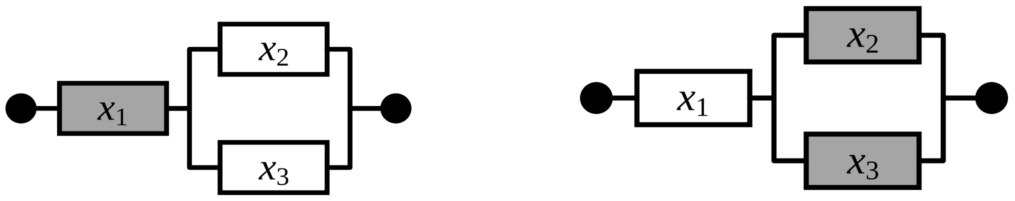


Fig. 1.5 Minimal Cut Sets (grey) of a series-parallel system

Similarly, the system has two MPSs $\{1,2\}$ and $\{1,3\}$ with corresponding state vectors $(1,1,0)$ and $(1,0,1)$. If any of the 1 elements would decrease to 0, the system would stop being operational. Fig. 1.6 shows the MPSs in the RBDs using the white color for the elements of the set.

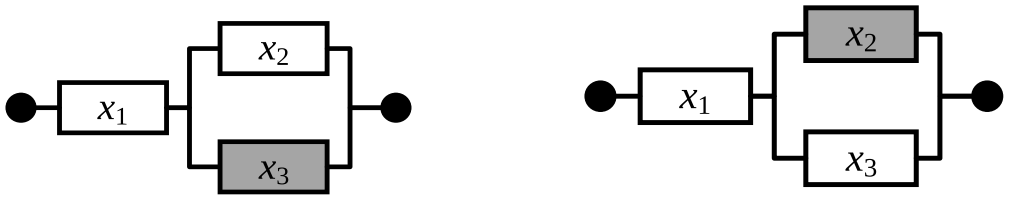


Fig. 1.6 Minimal Path Sets (white) of a series-parallel system

Finally, let us consider the state vector $(1,1,1)$, which corresponds to the path set $\{1,2,3\}$. The system in the state described by the set is operational. If we remove component 2 from the set, the system would still be operational. Therefore, the set is not an MPS.

1.3.3 K-out-of-n Systems

K-out-of-n system consists of n components and is functioning if at least k components are functioning. It is one of the common system types that we encounter in practice in areas such as software and hardware engineering [25]. The nature of the system is ideal for providing redundancy and therefore increasing fault tolerance [26] of the system. For example, if a k-out-of-n system serves as a subsystem of some bigger system, it can operate even when some of its components fail and therefore provide the time needed to either repair or replace failed components.

1.3.4 Complex Systems

A considerable number of systems that we encounter in practice are so-called complex systems. The complexity may have distinct causes for different system types. One of the more obvious properties is the number of components. For example, we consider a series-parallel system with a substantial number of components [24] to be a complex system. In the analysis of the system, we need to put a great emphasis on the efficient representation of the structure function. Moreover, various noncoherent systems are also complex. In this case, the reason for the complexity is that certain algorithms used in reliability analysis assume coherency and therefore are not applicable. An example of a system type from this category is logical circuits [14], especially with a higher number of gates. Finally, the complexity may also originate from the different nature of components of the system, which is often a case for nonhomogeneous MSS. We may encounter such systems in the analysis of healthcare systems [27] where different elements such as humans, software, and hardware interoperate within a single system. A challenging task in the analysis of such systems is the development of algorithms and a suitable representation for the system.

1.4 Topological Analysis

The structure function captures the topology of the system allowing us to perform a topological analysis of the system, which we can subsequently use to compare systems with different topologies. This sort of analysis can be useful in the process of system design. A basic measure that we use to compare topologies of MSS is the relative frequency of a system state j [6]:

$$Fr^{=j} = \mathrm{TD}(\phi(\boldsymbol{x}) \leftrightarrow j), \tag{1.9}$$

where $\phi(\boldsymbol{x})$ is structure function, $j \in \{0,1,\dots,m-1\}$ where m is the number of system states and $\mathrm{TD}(.)$ denotes truth density of a Boolean-valued function i.e., the relative number of state vectors for which the function takes value 1. Notice that the structure function $\phi(\boldsymbol{x})$ of an MSS is not a Boolean-valued function. To transform the function into Boolean-valued form we use the logical biconditional $\leftrightarrow$ defined as follows:

$$\phi(\boldsymbol{x}) \leftrightarrow j = \begin{cases} 1 & \phi(\boldsymbol{x}) = j \\ 0 & \text{otherwise.} \end{cases} \tag{1.10}$$

Relative frequency can also include multiple states in the form of the relative frequency of system states greater than j defined as [6]:

$$Fr^{\geq j} = \mathrm{TD}(\phi(\boldsymbol{x}) \geq j) = \sum_{h=j}^{m-1} Fr^{h}, \tag{1.11}$$

where $j \in \{1,2,\dots,m-1\}$. The argument of the truth density is defined using the logical biconditional as:

$$\phi(\boldsymbol{x}) \geq j = \bigvee_{h=j}^{m-1} (\phi(\boldsymbol{x}) \leftrightarrow h) = \begin{cases} 1 & \phi(\boldsymbol{x}) \geq j \\ 0 & \text{otherwise} \end{cases}, \tag{1.12}$$

where the $\vee$ operator denotes logical disjunction.

Notice that for BSS we do not need to transform the structure function using the logical biconditional since the structure function of a BSS is a Boolean function i.e., we can directly calculate $Fr^{=1} = \mathrm{TD}\big(\phi(\boldsymbol{x})\big)$ and $Fr^{=0} = 1 - Fr^{=1}$.

1.5 Probabilistic Analysis

The topological analysis considers only the topology of a system. It assumes that each state of a component is equally probable, which however is often not the case. Some components are more reliable than others, which influences behavior and consequently the reliability of the system. Therefore, in order to describe and analyze such systems more precisely, we need to use probabilistic analysis, which considers *component state probabilities*. These probabilities can be either time-independent or time-dependent and thus we also differentiate time-independent or time-dependent probabilistic analysis.

1.5.1 Time-independent Analysis

1.5.1.1 Description of States

We denote time-independent probabilities of BSS component states as:

$$\begin{aligned} p_i &= \Pr\{x_i = 1\}, \\ q_i &= \Pr\{x_i = 0\}, \end{aligned} \tag{1.13}$$

where $i = 1,2,\dots,n$. p_i, which is a probability that ith component is functioning is known as component reliability, and similarly, a probability that ith component failed q_i is known as component unreliability. Similarly, for an MSS we denote component state probability as:

$$p_{i,k} = \Pr\{x_i = k\}, \tag{1.14}$$

where $i = 1,2,\dots,n$ and $k = 0,1,\dots,m_i - 1$.

1.5.1.2 System Availability

Structure function and component state probabilities allow us to calculate global system characteristics known as system *availability* and *unavailability*. Availability of BSS agrees with the probability that the system is in state 1 and is defined as follows [1], [2]:

$$A(\boldsymbol{p}) = \Pr\{\phi(\boldsymbol{x}) = 1\}, \tag{1.15}$$

where $\boldsymbol{p} = (p_1, p_2, \dots, p_n)$ is a vector of component reliabilities. The unreliability of a BSS, which is a complementary measure for the availability and agrees with the probability that the system is in state 0, is defined as follows [1], [2]:

$$U(\boldsymbol{q}) = \Pr\{\phi(\boldsymbol{x}) = 0\}, \tag{1.16}$$

where $\boldsymbol{q} = (q_1, q_2, \dots, q_n)$ is a vector of component unreliabilities.

To generalize measures of availability and unavailability for MSS we consider two sets of system states. The first set contains states worse than state j and the second set contains state j and all better states. States in the second set represent the acceptable performance of the system for a particular use case and the state j represents the boundary state. Then, we define the availability of MSS with respect to state j, which agrees with the probability that the system is in state j or better, as follows [3], [4], [5]:

$$A^{\geq j}(\boldsymbol{p}) = \Pr\{\phi(\boldsymbol{x}) \geq j\}, \tag{1.17}$$

for $j = 1,2, \dots, m-1$. Similarly, we define the unavailability of MSS with respect to state j, which agrees with the probability that the system is in a state worse than j as [3], [4], [5]:

$$U^{\geq j}(\boldsymbol{p}) = \Pr\{\phi(\boldsymbol{x}) < j\}, \tag{1.18}$$

for $j = 1,2, \dots, m-1$. Lastly, let us notice that in the case of an MSS, the vector $\boldsymbol{p}$ is actually a matrix $\mathbb{P}_{n,\max(m_i)}$ for $i = 1,2, \dots, n$ – where $p_{i,k}$ denotes an element of the matrix. However, for the consistency with the literature, we keep the notation $A^{\geq j}(\boldsymbol{p})$.

Measures of system availability and unavailability allow us to examine system reliability taking into account not only the topology captured by the structure function but also the probabilities of component states. This allows us to compare not only different system topologies but also investigate how the reliabilities of individual components influence the overall availability of the system.

In addition to system availability and unavailability, we can also define probabilities of individual system states i.e., a probability that a system is in state j [28], which we denote as $\Pr\{\phi(\boldsymbol{x}) = j\}$.

The system states probabilities, availability, and unavailability are closely tied, and we can compute one in terms of the other using the following formulas [28]:

$$\Pr\{\phi(\boldsymbol{x}) = j\} = \begin{cases} 1 - A^{\geq 1}(\boldsymbol{p}) & \text{if } j = 0 \\ A^{\geq j}(\boldsymbol{p}) - A^{\geq j+1}(\boldsymbol{p}) & \text{if } j \in \{1,2, \dots, m-2\}, \\ A^{\geq m-1}(\boldsymbol{p}) & \text{if } j = m-1 \end{cases} \tag{1.19}$$

$$A^{\geq j} = \sum_{h=j}^{m-1} \Pr\{\phi(\boldsymbol{x}) = j\}, U^{\geq j} = \sum_{h=0}^{j-1} \Pr\{\phi(\boldsymbol{x}) = j\}, \tag{1.20}$$

for $j \in \{1,2,\dots,m-1\}$. Another important characteristic of a system is expected system performance. Most of the time, the numbers that we use to describe system states are abstract i.e., they do not have a physical meaning. Expected system performance allows us to describe a system in terms of physical performance using the following definition [28]:

$$O(\boldsymbol{p}) = \sum_{j=0}^{m-1} o_j \Pr\{\phi(\boldsymbol{x}) = j\} = o_0 + \sum_{j=1}^{m-1} \left(o_j - o_{j-1}\right) A^{\geq j}(\boldsymbol{p}), \tag{1.21}$$

where o_j denotes physical performance associated with system state j. As an example, let us consider a transportation network that can operate in 4 possible states denoted by numbers 0,1,2, and 3. Physical performance associated with the states are the amounts $0, 100, 200,$ and 300 of units respectively that the network can transport. So, for example, if the system is in state 2 it can transport 200 units. Therefore, if there is a requirement that the network must be able to transport at least 170 units we need to calculate system availability with respect to state 2 since it is the first state that satisfies the requirements.

1.5.2 Time-dependent Analysis

1.5.2.1 Description of States

A limitation of the time-independent analysis is the assumption that the component state probability is a constant. However, if we observe a component of a real-world system, we would notice that the component state probability evolves. Specifically, the component state probability usually deteriorates in time. This can be intuitively explained, for example, by physical wear and tear of technical components.

Since we describe the system as an MSS, we consequently need to describe the change of state of the ith component in time. For such a description, we use a state function $z_i(t)$. Unfortunately, the number of ways in which the state of a component can change is infinite. For instance, let us consider four specific state functions of a 3-state component depicted in Fig. 1.7. Each chart in the figure describes a possible evolution of the component state in time.

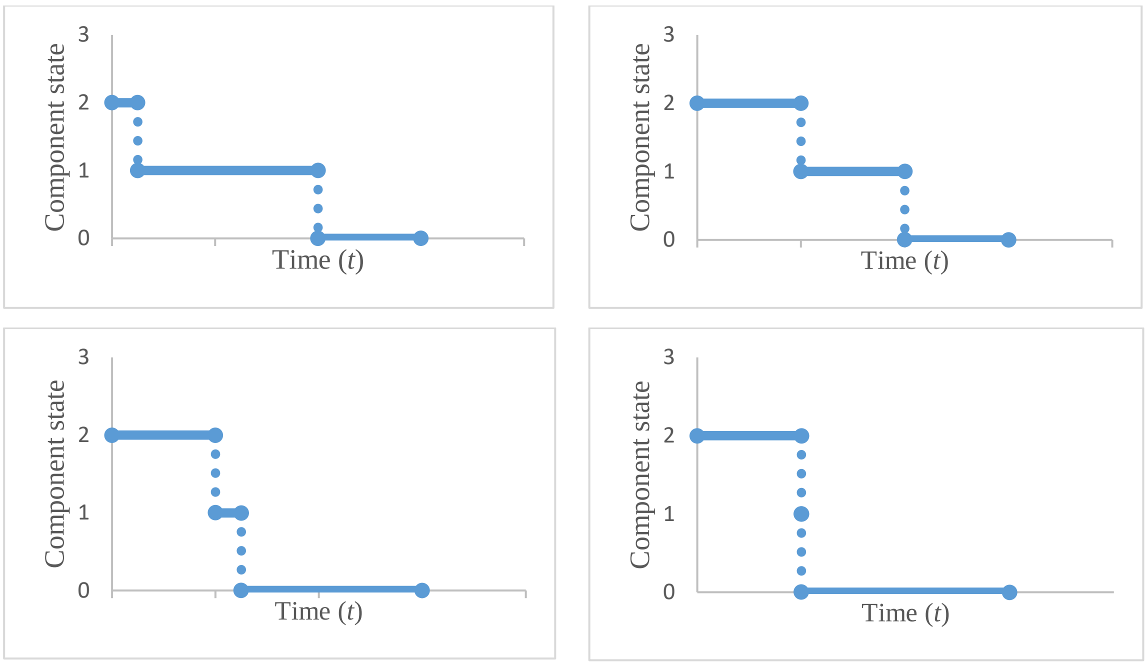


Fig. 1.7 Different state functions modeling the behavior of a 3-state component

The set of values of each state function is $\{0,1,\dots,m_i-1\}$ in the most general case of nonhomogeneous MSS. If we consider all possible functions at time t, then the proportion of functions that take value s agrees with the probability that ith component is in state s. Consequently, we can introduce a discrete random variable Z_i describing all states of the ith component [3], [29], [30]:

$$p_{i,s}=\Pr\{Z_i=s\}, s=0,1,\dots,m_i-1, \\ \sum_{s=0}^{m_i-1}p_{i,s}=1 \tag{1.22}$$

Since random variable Z_i changes in time we can define a function of time $Z_i(t)$ and then define a stochastic process as a collection of random variables [5], [31]:

$$\{Z_i(t);t\geq 0\}. \tag{1.23}$$

Finally, we define time-dependent reliability and unreliability of BSS component using the function of time as:

$$p_i(t)=\Pr\{Z_i(t)=1\}, \\ q_i(t)=\Pr\{Z_i(t)=0\}, \\ p_i(t)+q_i(t)=1, t>0. \tag{1.24}$$

And time-dependent MSS's component state probabilities as:

$$p_{i,s}(t)=\Pr\{Z_i(t)=s\}, s=0,1,\dots m_i-1, \tag{1.25}$$

$$\sum_{s=0}^{m_i-1} p_{i,s}(t) = 1, t \geq 0.$$

Subsequently, we can use time-dependent component state probabilities along with the structure function to define the system state function [1], [5]:

$$\boldsymbol{Z}(t) = \phi\big(Z_1(t), Z_2(t), \dots, Z_n(t)\big) = \phi\big(\boldsymbol{Z}(t)\big), \tag{1.26}$$

where $\boldsymbol{Z}(t)$ is a vector of component state functions.

1.5.2.2 System Availability and Reliability

Reliability is one of the basic time-dependent characteristics of a BSS. It is defined as the probability that the system operates without failure in the time interval $(0, t\rangle$ given that the system was functioning at time $t = 0$:

$$R(t) = \Pr\{T_f > t\}; R(0) = 1, \tag{1.27}$$

where T_f is a random variable representing time to failure [1]. A complementary characteristic to reliability is system unreliability, which represents the probability that the system will fail in the time interval $(0, t\rangle$ given that it was functioning at time $t = 0$:

$$F(t) = \Pr\{T_f \leq t\}; F(0) = 1. \tag{1.28}$$

As time progresses the reliability of the system degrades inevitably leading to a failure. Maintainability of a BSS is the ability of the system to be maintained in or restored to an acceptable state. Mathematically, we describe maintainability using a function that defines the probability that the system maintenance will be performed in a specific period:

$$M(t) = \Pr\{T_m \leq t\}, \tag{1.29}$$

where T_m is a random variable identifying the time needed for system maintenance. The exact meaning of the random variable depends on the type of maintenance that can either be preventive or corrective [1]. Furthermore, the ability to be repaired implies that there exist two types of systems repairable and non-repairable while maintainability describes only the repairable systems. The characteristics of reliability and maintainability can be combined into the availability and unavailability of BSS, which we also introduced in the case of time-independent analysis. The time-dependent system availability and unavailability are defined using the system state function as [1]:

$$A(t) = \Pr\{\phi\big(\boldsymbol{Z}(t)\big) = 1\}, \tag{1.30}$$

$$U(t) = \Pr\{\phi\big(\boldsymbol{Z}(t)\big) = 0\} = 1 - A(t). \tag{1.31}$$

We can generalize system availability and unavailability using the same rationale as in the time-independent analysis by splitting the systems state into two sets of acceptable and unacceptable states where the state j is the first acceptable state. Then we define the time-dependent availability of MSS with respect to state j as [3], [5]:

$$A^{\geq j}(t) = \Pr\{\phi(\boldsymbol{Z}(t)) \geq j\} = \sum_{h=j}^{m_i-1} \Pr\{\phi(\boldsymbol{Z}(t)) = h\}, \tag{1.32}$$

and time-dependent system unavailability of MSS with respect to state j as [3], [5]:

$$U^{\geq j}(t) = \Pr\{\phi(\boldsymbol{Z}(t)) < j\} = \sum_{h=0}^{j-1} \Pr\{\phi(\boldsymbol{Z}(t)) = h\} = 1 - A^{\geq j}(t). \tag{1.33}$$

We can see that a principal difference between time-independent and time-dependent probabilistic analysis is that characteristics such as system availability or component state probabilities are functions of time in the latter case instead of being a single number in the former case.

1.6 Importance Analysis

Topological and probabilistic analysis provide means of studying systems using system characteristics such as system state frequency or system availability. Though we can use the characteristics to study how for example a change in the component availability affects the overall availability of the system, the characteristics nevertheless describe the entire system. For quantification of the influence of individual components, we use characteristics known as Importance Measures (IMs). The literature presents various IMs. In this section, we briefly introduce the commonly used ones.

1.6.1 Structural Importance

Structural Importance (SI) represents one of the simplest IMs. It considers only the topology of the system and is part of the topological analysis. For a BSS SI is defined as follows [7]:

$$\mathrm{SI}_i = \frac{\sum_{(\cdot_i,\boldsymbol{x})\in\{0,1\}^{n-1}}\big(\phi(1_i,\boldsymbol{x}) - \phi(0_i,\boldsymbol{x})\big)}{2^{n-1}}, \tag{1.34}$$

where $(\cdot_i, \boldsymbol{x})$ is a state vector without ith component, $\{0,1\}^{n-1}$ represents all possible state vectors of the form $(\cdot_i, \boldsymbol{x})$ and $(a_i, \boldsymbol{x}) = (x_1, x_2, \dots, x_{i-1}, a, x_{i+1}, \dots, x_n)$. It agrees with the relative number of situations when the ith component is critical for the system activity i.e. when the state of the component decides whether the system is functioning or failed. Since the SI considers only the topology, it is useful in situations when we do not have information about component reliabilities.

The SI measure can also be generalized to describe components of MSS. However, multiple components and system states allow different interpretations of the measure. Therefore, the authors proposed several definitions. A definition similar to (1.34) considering a specific change in the component state can be found in [32].

1.6.2 Birnbaum's Importance

One of the limitations of the SI is that it does not consider component state probabilities. Birnbaum's importance (BI) [33] considers system topology as well as component state probabilities. The literature offers multiple ways of calculating BI for BSS [7], [34], [35]:

$$\mathrm{BI}_i = \Pr\{\phi(1_i, \boldsymbol{x}) - \phi(0_i, \boldsymbol{x}) > 0\} \tag{1.35}$$

$$= \frac{\partial A(\boldsymbol{p})}{\partial p_i} = \frac{\partial U(\boldsymbol{p})}{\partial q_i} \tag{1.36}$$

$$= A(1_i, \boldsymbol{p}) - A(0_i, \boldsymbol{p}) = U(0_i, \boldsymbol{q}) - U(1_i, \boldsymbol{q}), \tag{1.37}$$

where $A(1_i, \boldsymbol{p}), A(0_i, \boldsymbol{p})$ denotes system availability if ith component is always functioning or failed respectively and similarly $U(1_i, \boldsymbol{p}), U(0_i, \boldsymbol{p})$ denotes system unavailability if ith component is always functioning or failed respectively.

In [7] the authors present several meanings of the definitions (1.35) – (1.37). Definition (1.35) agrees with the probability that failure (repair) of the ith component coincides with failure (repair) of the system. Definition (1.36) defines BI in terms of the rate at which system availability (unavailability) improves (degrades) with the reliability (unreliability) of the component. Lastly, according to definition (1.37), BI describes the decrease in system availability if ith component fails or similarly a decrease of system unavailability if ith component is repaired.

As with the SI, the authors proposed several generalizations of BI for MSS. A straightforward generalization that considers a specific change in a component state with respect to system state j following the definition (1.37) is presented in [32] and a more general version that incorporates multiple changes of a component state can be found in [7]. Also, a different approach for homogeneous MSS can be found in [28].

1.6.3 Criticality Importance

One of the drawbacks of BI is that it does not consider the current value of the component availability/unavailability. For example, the component might be highly influential according to BI but the probability that the component will fail might be very small, which practically reduces the importance of the component since it is almost always functioning. Criticality importance (CI) is defined in terms of BI aiming to solve the drawback by including component availability/unavailability in the calculation. There exist two versions of CI one defined in terms of system functioning (C$_s$I) and the other one defined in terms of system failure (C$_f$I). For a BSS the definitions have the following form [7]:

$$\mathrm{C}_f\mathrm{I}_i = \mathrm{BI}_i \frac{q_i}{U(\boldsymbol{q})}, \tag{1.38}$$

$$\mathrm{C}_s\mathrm{I}_i = \mathrm{BI}_i \frac{p_i}{A(\boldsymbol{q})}. \tag{1.39}$$

C$_f$I measures the probability that the system failed because of the ith component given that the system failed and C$_s$I measures the probability that ith component is critical for the system functioning given that the system is functioning. Since the CI is defined in terms of BI its generalization for MSS follows the generalized BI using availability or unavailability with respect to a given system state j.

1.6.4 Fussell-Vesely's Importance

A component contributes to the failure of a system when the failure of the component causes at least one MCS containing the component to fail. Therefore, a measure known as Fussell-Veseley's importance (FVI), which for a BSS is defined in terms of MCSs as follows [7]:

$$\mathrm{FV}_c\mathrm{I}_i = \frac{\Pr\{\mathrm{MCSs}(i)\}}{U(\boldsymbol{q})}, \tag{1.40}$$

where the notation $\{\mathrm{MCSs}(i)\}$ represents the event that at least one MCS containing component i has failed. On the other hand, the component also contributes to the system's functioning. Therefore, another way to define FVI for a BSS is in terms of MPSs [7], [36]:

$$\mathrm{FV}_p\mathrm{I}_i = \frac{\Pr\{\mathrm{MPSs}(i)\}}{A(\boldsymbol{p})}, \tag{1.41}$$

where the notation $\{\mathrm{MPSs}(i)\}$ represents the event that at least one MPS containing component i is functioning. The literature offers several generalizations of FVI [37], [38], [39] for MSS, however, note that their meaning does not correspond to the FVI defined for BSS since the generalizations are not based on MCSs but use different approaches.

2 Discrete Function

In section 1.2, we introduced multiple definitions of the structure function (1.1), (1.2), and (1.3). Each definition has a form of a discrete function. As we also showed, the structure function is an integral part of the topological analysis, probabilistic analysis, and calculations of various importance measures. Therefore, discrete functions are one of the fundamental tools for reliability analysis. Thus, this chapter focuses on the introduction of discrete function types that are relevant for the reliability analysis. Also, it deals with the analysis of properties of discrete functions, which we can subsequently interpret as properties of the described system. Finally, the chapter presents several approaches to the representation of the discrete function. Since we aim at the analysis of complex systems with a high number of components, we mainly focus on the efficiency of the representation. At the end of the chapter, we introduce decision diagrams as a suitable representation of the structure function in the analysis of complex systems.

2.1 Discrete Function Types

In general, a function is a mapping from a domain to a codomain. A discrete function is a function in which each variable takes values from a finite set – the domain of the variable. The domain of a discrete function itself is a Cartesian product of the domains of all its variables. We denote the elements of the domain using vector notation $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ where n is the number of variables and x_i denotes ith variable. We sometimes refer to $\boldsymbol{x}$ as the input vector. The codomain of a discrete function is also finite. In general, a discrete function is defined as the following mapping [10]:

$$f(\boldsymbol{x}) \colon \times_{i=1}^{n} \mathcal{D}_i \to \mathcal{L}, \tag{2.1}$$

where the operator $\times$ denotes the Cartesian product and n is the number of variables. The sets $\mathcal{D}_i$ for $i = 1,2,\dots,n$ are domains of variables and $\mathcal{L}$ is the codomain of the function (set of values of the function). Note that the sets $\mathcal{D}_i$ and $\mathcal{L}$ are finite but not necessarily of the same cardinalities. Depending on the cardinalities of the sets $\mathcal{D}_i$ and $\mathcal{L}$, we recognize different types of discrete functions whose description follows.

2.1.1 Boolean Function

The simplest and the most well-known type of discrete function is the Boolean function. Variables of the Boolean function take values from the set $\{0,1\}$, where the values 0 and

1 are often denoted as false and true respectively. The definition (2.1) simplifies to the following form [8]:

$$f(\boldsymbol{x}) : \times_{i=1}^{n} \{0,1\} \rightarrow \{0,1\}. \qquad (2.2)$$

Let us note that the values 0 and 1 can be interpreted differently such as “off–on” or “failed–working”, depending on the application of the Boolean function.

2.1.2 Multiple-Valued Logic Function

The Multiple-Valued Logic (MVL) function can be seen as a generalization of the Boolean function where, instead of just two, the MVL variables take values from the set $\{0,1,\dots,m-1\}$, where $m \in \mathbb{N}$ and $m > 1$. The codomain of the MVL function is also the set $\{0,1,\dots,m-1\}$. For the MVL function, the definition (2.1) has the following form [9]:

$$f(\boldsymbol{x}) : \times_{i=1}^{n} \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}. \qquad (2.3)$$

Finally, let us note that in some places in the thesis, we denote the MVL function using the notation $f_m(\boldsymbol{x})$ to better express the number of values m.

2.1.3 Integer Function

The integer function further generalizes the MVL function in a way that the domain of each variable and the codomain of the function are allowed to have different cardinalities. The definition of the integer function has the following form [10]:

$$f(\boldsymbol{x}) : \times_{i=1}^{n} \{0,1,\dots,m_i-1\} \rightarrow \{0,1,\dots,m-1\}, \qquad (2.4)$$

where m_i for $i = 1,2,\dots,n$ and $m, m_i \in \mathbb{N}$ and $m > 1$ and $m_i > 1$. To explicitly include the domains in the notation, we denote the integer function as $f_{m;m_1,m_2,\dots,m_n}(\boldsymbol{x})$ in some places. Notice that the definition (2.4) is almost identical to the general definition of a discrete function (2.1). However, an important difference is that the definition (2.4) specifies that elements of the variable domains and codomain are natural numbers starting from 0 up to $m_i - 1$ whereas the general definition (2.1) only specifies that the sets must be finite.

2.1.4 Pseudo-logic Function

The flexibility of the definition (2.4) allows for different special cases. Both Boolean function and MVL function can be considered as special cases of the integer function. Moreover, there exists another special case which is the pseudo-logic function. The important property of the pseudo-logic function is that its codomain is identical to the codomain of the Boolean function. The pseudo-logic function has the following definition [10]:

$$f(\boldsymbol{x}): \times_{i=1}^{n} \{0,1,\dots,m_i - 1\} \rightarrow \{0,1\}. \tag{2.5}$$

We also sometimes refer to it as a Boolean-valued integer function. Boolean-valued integer functions usually emerge when we need to identify elements of the domain of the function that satisfy some property. For example, let $f(\boldsymbol{x})$ be an integer function and let $g(\boldsymbol{x}) = f(\boldsymbol{x}) > 1$. Then $g(\boldsymbol{x})$ is a Boolean-valued integer function that evaluates to 1 for $\boldsymbol{x}$ at which the function $f(\boldsymbol{x})$ evaluates to a number greater than 1.

2.2 Discrete Function Analysis

The analysis of functions of real and complex variables is an established task in mathematics. Over the years, numerous methods that use tools of differential calculus to analyze their dynamic properties have been introduced. The literature offers similar tools for the analysis of discrete functions as well. Before we proceed with the description of the tools, we need to introduce the cofactor of a discrete function.

2.2.1 Discrete Function Cofactor

We define the cofactor for the integer function since it is the most general form of the discrete function that we consider in the thesis. Let $f(\boldsymbol{x}) = f(x_1, x_2, \dots, x_n)$ be an integer function. Then its cofactor with respect to the variable x_i and value a is:

$$f(a_i, \boldsymbol{x}) = f(x_1, x_2, \dots, x_{i-1}, a, x_{i+1}, \dots, x_{n-1}, x_n), \tag{2.6}$$

where $a \in \{0,1,\dots,m_i - 1\}$, $i \in \{1,2,\dots,n\}$ and m_i is the size of the domain of the i^{th} variable. The cofactor is a function of $n - 1$ variables, which simplifies the original function by setting the value of a variable to a constant a.

2.2.2 Logical Differential Calculus

Logical differential calculus [10], [40] is the mathematical approach that we use to analyze the dynamic properties of discrete functions. It offers various tools analogous to traditional differential calculus. The relevant tool for this thesis is a logic derivative. The derivative has different forms – we start by introducing the simplest ones concerning Boolean functions.

2.2.2.1 Boolean Derivative

The Boolean derivative is a basic type of logic derivative and, as the name suggests, we can apply it to Boolean functions. To define the derivative with respect to the variable x_i, we use the cofactor of Boolean function and logical exclusive disjunction (XOR) Boolean operator:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = f(0_i, \boldsymbol{x}) \oplus f(1_i, \boldsymbol{x}). \tag{2.7}$$

Alternatively, we can also use a different notation with the same meaning:

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} = \begin{cases} 1, & \text{if } f(0_i, \boldsymbol{x}) \neq f(1_i, \boldsymbol{x}) \\ 0, & \text{otherwise} \end{cases}. \tag{2.8}$$

The derivative defined as (2.7) or (2.8) is a Boolean function of $n - 1$ variables. It describes properties of the original function in a way that it evaluates to 1 for elements of the domain where the change (either from 0 to 1 or from 1 to 0) of the value of ith variable causes a change of the value of the original function (again either from 0 to 1 or from 1 to 0).

Since XOR is a symmetrical operation (it does not depend on the order of its arguments) the derivative creates pairs of vectors of the domain of the form $(0_i, \boldsymbol{x}) = (x_1, x_2, \dots, x_{i-1}, 0, x_{i+1}, \dots, x_n)$ and $(1_i, \boldsymbol{x}) = (x_1, x_2, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_n)$. If the change of the value of ith variable in the first vector causes a change in the value of the function, then the opposite change in the second vector necessarily causes the opposite change in the value of the function. A disadvantage of this derivative is that it hides information about how (in which direction) the value of the variable changed and in which direction the value of the function changed. Therefore, to describe the properties of the function more precisely we use directional Boolean derivatives.

2.2.2.2 Directional Boolean Derivative

In the general form, we define directional Boolean derivative as [10], [20]:

$$\frac{\partial f(j \to \bar{j})}{\partial x_i(s \to \bar{s})} = \begin{cases} 1, & \text{if } f(s_i, \boldsymbol{x}) = j \text{ and } f(\bar{s}_i, \boldsymbol{x}) = \bar{j} \\ 0, & \text{otherwise} \end{cases}, \tag{2.9}$$

where $j, s \in \{0,1\}$. Derivative (2.9) is a function of $n - 1$ variables that evaluates to 1 for the vectors of the form $(s_i, \boldsymbol{x})$ for which the function evaluates to j if it holds that the function evaluates to $\bar{j}$ when the value of the ith variable changes from s to $\bar{s}$. Definition (2.9) allows four specific types of the derivative for four possible combinations of values of s and j. In first two cases s and j have the same value, which means that a change of the value of ith variable results in the same change in the value of the function. This type of directional derivative is known as Direct Partial Boolean Derivative (DPBD) and is defined as [10], [20]:

$$\frac{\partial f(1 \to 0)}{\partial x_i(1 \to 0)} = \frac{\partial f(0 \to 1)}{\partial x_i(0 \to 1)} = f(1_i, \boldsymbol{x}) \wedge \overline{f(0_i, \boldsymbol{x})}, \tag{2.10}$$

where ∧ denotes logical conjunction. Notice that the definition suggests that both derivatives are represented by the same function. However, the key difference is in the domain of the functions. Unlike Boolean derivative (2.7), which is defined for all elements of the domain of the original function, the DPBD is only defined for elements of the form $(1_i, \boldsymbol{x})$ for derivative $\partial f(1 \to 0)/\partial x_i(1 \to 0)$ and for elements of the form $(0_i, \boldsymbol{x})$ for derivative $\partial f(0 \to 1)/\partial x_i(0 \to 1)$. Therefore, the derivatives must be computed in the specified points to retain their meaning.

The two other cases include situations when s and j have opposite values i.e., when the change of the value of the ith variable causes the inverse (hence the name) change in the value of the function. This type of direct Boolean derivative is known as Inverse Partial Boolean Derivative (IPBD) and is defined as [10]:

$$\frac{\partial f(1 \to 0)}{\partial x_i(0 \to 1)} = \frac{\partial f(0 \to 1)}{\partial x_i(1 \to 0)} = f(0_i, \boldsymbol{x}) \wedge \overline{f(1_\iota, \boldsymbol{x})}. \tag{2.11}$$

Just like with DPBDs, both derivatives are represented by the same function. The difference is again in the domain where IPBD is only defined for vectors of the form $(1_i, \boldsymbol{x})$ for derivative $\partial f(0 \to 1)/\partial x_i(1 \to 0)$ and for elements of the form $(0_i, \boldsymbol{x})$ for derivative $\partial f(1 \to 0)/\partial x_i(0 \to 1)$. Fig. 2.1 shows a summary of the four types of direct Boolean derivatives. We can see that each derivative is evaluated only for the elements where ith variable has the value from which we observe its change.

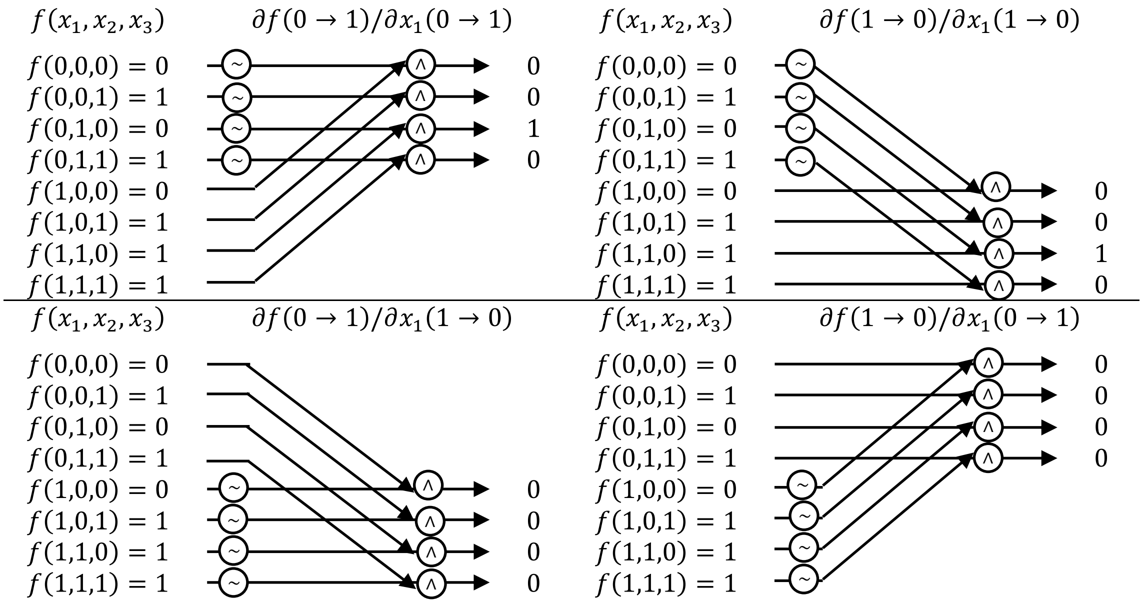


Fig. 2.1 Four possible types of directional Boolean derivatives of the function $f(\boldsymbol{x}) = x_1 x_2 \vee x_3$ depicted using flow diagrams where ∼ denotes logical negation and ∧ denotes logical conjunction

2.2.2.3 Directional Logic Derivative

The directional logic derivative is a generalization of the directional Boolean derivative for the MVL function. Let $f_m(\boldsymbol{x})$ be an MVL function. Then we define directional logic derivative with respect to variable x_i as [10]:

$$\frac{\partial f_m(j \to h)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{if } f_m(s_j, \boldsymbol{x}) = j \text{ and } f_m(r_j, \boldsymbol{x}) = h \\ 0, & \text{otherwise} \end{cases}, \tag{2.12}$$

where $s, r, j, h \in \{0,1, \dots, m-1\}, s \neq r$ and $j \neq h$. The derivative is a function of $n-1$ m-valued variables, which evaluates to 1 for elements of the domain where i^{th} variable has value s, the function evaluates to j and it holds that if the value of the variable changes from s to r the value of the function changes to h. Notice that the derivative is a pseudo-logic function that we have described in section 2.1.4.

Depending on the relations of the values s, r and j, h we recognize two types of directional logic derivatives. The first type is Direct Partial Logic Derivative (DPLD) for which it holds that either $s > r$ and $j > h$ or $s < r$ and $j < h$ i.e., a specific increase (decrease) of the value of the variable causes a specific increase (decrease) of the value of the function. The second type is Inverse Partial Logic Derivative (IPLD) for which it holds that either $s > r$ and $j < h$ or $s < r$ and $j > h$ i.e., a specific increase (decrease) of the value of the variable causes a specific decrease (increase) of the value of the function.

Analogously to DPBDs, the following relation holds for directional logic derivatives:

$$\frac{\partial f_m(j \to h)}{\partial x_i(s \to r)} = \frac{\partial f_m(h \to j)}{\partial x_i(r \to s)}. \tag{2.13}$$

Although derivatives in both directions are represented by the same function the key difference is again in the domains since the directional logic derivative is only defined for the vectors of the form $(s_i, \boldsymbol{x})$ and $(r_i, \boldsymbol{x})$ for DPLD and IPLD respectively.

We can also use the directional logic derivative to analyze an integer function. We define it in an almost identical way just by changing the type of the function [30]:

$$\frac{\partial f_{m;m_1,m_2,\dots,m_n}(j \to h)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{if } f_{m;m_1,m_2,\dots,m_n}(s_i, \boldsymbol{x}) = j \text{ and } f_{m;m_1,m_2,\dots,m_n}(r_i, \boldsymbol{x}) = h \\ 0, & \text{otherwise} \end{cases}, \tag{2.14}$$

where $s, r \in \{0,1, \dots, m_i\}, s \neq r \text{ and } j, h \in \{0,1, \dots, m\}, j \neq h.$ Just like with MVL functions, we recognize two types of derivatives, which are DPLD and IPLD. We can see that the directional logic derivative (2.14) for the integer function is the most general form that includes directional derivatives of the MVL function and also directional Boolean derivatives. Therefore, we will consider just this most general case in the rest of the section.

Definition (2.12) allows $m^2 * (m-1)^2$ specific directional derivatives of MVL function for all possible combinations of values of j, h, s, r and definition (2.14) allows $m * (m-1) * m_i * (m_i - 1)$ specific directional derivatives of integer function. We can see that even for smaller values of m the number of possible derivatives is considerable. However, often we are not interested in the exact influence of a specific change, but we want, for example, to know whether a certain change in the value of a variable causes any change in the value of the function. Therefore, to obtain a better overall characteristic of the examined function we need to use a different type of logic derivative known as integrated directional logic derivatives.

2.2.2.4 Integrated Directional Logic Derivative

The literature recognizes three types of integrated directional logic derivatives. Each of them contains information that is contained in several simple directional logic derivatives.

2.2.2.4.1 IDPLD of type I

Integrated logic derivative of type I describes situations when the change of the value of the ith variable from s to r causes a change in the value of the function:

- from the value j to a value less than j,
- from a value less than j to the value j,
- from a value greater than j to the value j,
- from the value j to a value greater than j.

The above-enumerated possibilities of a change suggest that the derivative can be defined in four configurations [30]:

$$\begin{aligned}\frac{\partial f_{m;m_1,\dots,m_n}(j \searrow)}{\partial x_i(s \to r)} &= \bigvee_{h=0}^{j-1} \frac{\partial f_{m;m_1,\dots,m_n}(j \to h)}{\partial x_i(s \to r)} \\ &= \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) = j \text{ and } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) < j \\ 0, & \text{otherwise} \end{cases},\end{aligned} \tag{2.15}$$

$$\begin{aligned}\frac{\partial f_{m;m_1,\dots,m_n}(\nearrow j)}{\partial x_i(s \to r)} &= \bigvee_{h=0}^{j-1} \frac{\partial f_{m;m_1,\dots,m_n}(h \to j)}{\partial x_i(s \to r)} \\ &= \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) < j \text{ and } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) = j \\ 0, & \text{otherwise} \end{cases},\end{aligned} \tag{2.16}$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(\searrow j)}{\partial x_i(s \to r)} = \bigvee_{h=j+1}^{m-1} \frac{\partial f_{m;m_1,\dots,m_n}(h \to j)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) > j \text{ and } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) = j \\ 0, & \text{otherwise} \end{cases}, \tag{2.17}$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(j \nearrow)}{\partial x_i(s \to r)} = \bigvee_{h=j+1}^{m-1} \frac{\partial f_{m;m_1,\dots,m_n}(j \to h)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) = j \text{ and } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) > j \\ 0, & \text{otherwise} \end{cases}, \tag{2.18}$$

where the symbol $\vee$ denotes logical disjunction. Notice that we can use the logical disjunction since the simple directional derivatives that we operate on are pseudo-logic functions. Derivatives (2.15) and (2.16) are defined for $s, r \in \{0,1, m_i - 1\}$ and $j \in \{1,2, \dots, m-1\}$ and derivatives (2.17) and (2.18) are defined for $s, r \in \{0,1, m_i - 1\}$ and $j \in \{0,1, \dots, m-2\}$. And finally, all of them should be computed only for elements of the form $(s_i, \boldsymbol{x})$. In summary the integrated directional logic derivative of type I identifies situations when a specific change of a value of a variable causes a change of the function from state j to a worse (better) state or vice versa.

2.2.2.4.2 IDPLD of type II

The integrated logic derivative of type II describes situations when the change of the value of the ith variable from s to r causes an improvement of the value of the function or in the second case a decrement of the value of the function. Therefore, we can define it in two configurations [30]:

$$\frac{\partial f_{m;m_1,\dots,m_n}(\searrow)}{\partial x_i(s \to r)} = \bigvee_{j=1}^{m-1} \frac{\partial f_{m;m_1,\dots,m_n}(j \searrow)}{\partial x_i(s \to r)} = \bigvee_{j=1}^{m-1} \bigvee_{h=0}^{j-1} \frac{\partial f_{m;m_1,m_2,\dots,m_n}(j \to h)}{\partial x_i(s \to r)}, = \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) > f_{m;m_1,\dots,m_n}(r_i, \boldsymbol{x}) \\ 0, & \text{otherwise} \end{cases}, \tag{2.19}$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(\nearrow)}{\partial x_i(s \to r)} = \bigvee_{j=1}^{m-1} \frac{\partial f_{m;m_1,\dots,m_n}(\nearrow j)}{\partial x_i(s \to r)}$$

$$= \bigvee_{j=1}^{m-1} \bigvee_{h=0}^{j-1} \frac{\partial f_{m;m_1,m_2,\dots,m_n}(h \to j)}{\partial x_i(s \to r)}, \tag{2.20}$$

$$= f(x) = \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i,\boldsymbol{x}) < f_{m;m_1,\dots,m_n}(r_i,\boldsymbol{x}) \\ 0, & \text{otherwise} \end{cases}.$$

Both versions of the derivative should be computed for elements of the form $(s_i,\boldsymbol{x})$. In summary, the integrated directional logic derivative of type II identifies situations when a specific change in the value of a variable causes any improvement (decrement) of the value of the function. Notice that we can define the derivative in terms of the derivatives of type I, which describe the change more precisely. Also, let us notice that the derivative of type II is the only type that does not contain logical and in its definition.

2.2.2.4.3 IDPLD of type III

Finally, the integrated logic derivative of type III describes situations when the change of the value of the ith variable from s to r causes a change in the value of the function in one of the following ways:

- from a value greater than or equal to j to a value less than j,
- from a value less than j to a value greater than or equal to j,
- from a value greater than j to a value less than or equal to j,
- from a value less or equal to j to a value greater than j.

$$\frac{\partial f_{m;m_1,\dots,m_n}(h_{\geq j} \to h_{<j})}{\partial x_i(s \to r)} = \bigvee_{h_u=j}^{m-1} \bigvee_{h_d=0}^{j-1} \frac{\partial f_{m;m_1,\dots,m_n}(h_u \to h_d)}{\partial x_i(s \to r)}$$

$$= \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i,\boldsymbol{x}) \geq j \text{ and } f_{m;m_1,\dots,m_n}(r_i,\boldsymbol{x}) < j \\ 0, & \text{otherwise} \end{cases}, \tag{2.21}$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(h_{<j} \to h_{\geq j})}{\partial x_i(s \to r)} = \bigvee_{h_d=0}^{j-1} \bigvee_{h_u=j}^{m-1} \frac{\partial f_{m;m_1,\dots,m_n}(h_d \to h_u)}{\partial x_i(s \to r)}$$

$$= \begin{cases} 1, & \text{if } f_{m;m_1,\dots,m_n}(s_i,\boldsymbol{x}) < j \text{ and } f_{m;m_1,\dots,m_n}(r_i,\boldsymbol{x}) \geq j \\ 0, & \text{otherwise} \end{cases}, \tag{2.22}$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(h_{>j} \rightarrow h_{\leq j})}{\partial x_i(s \rightarrow r)} = \bigvee_{h_u=j+1}^{m-1} \bigvee_{h_d=0}^{j} \frac{\partial f_{m;m_1,m_2,\dots,m_n}(h_u \rightarrow h_d)}{\partial x_i(s \rightarrow r)} \tag{2.23}$$
$$= \begin{cases} 1, & if\ f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) > j \text{ and } f_{m;m_1,\dots,m_n}(r_i, \boldsymbol{x}) \leq j \\ 0, & \text{otherwise} \end{cases},$$

$$\frac{\partial f_{m;m_1,\dots,m_n}(h_{\leq j} \rightarrow h_{>j})}{\partial x_i(s \rightarrow r)} = \bigvee_{h_d=0}^{j} \bigvee_{h_u=j+1}^{m-1} \frac{\partial f_{m;m_1,m_2,\dots,m_n}(h_d \rightarrow h_u)}{\partial x_i(s \rightarrow r)} \tag{2.24}$$
$$= \begin{cases} 1, & if\ f_{m;m_1,\dots,m_n}(s_i, \boldsymbol{x}) \leq j \text{ and } f_{m;m_1,\dots,m_n}(r_i, \boldsymbol{x}) > j \\ 0, & \text{otherwise} \end{cases},$$

where the notation $h_{\leq j}$ denotes all system states that are less or equal to j while the meaning is analogous for other relational operators.

2.3 Application of Logic Derivatives

The importance measures that we described in section 1.6 quantify how a change in a component state or reliability influences the state and reliability of the entire system. The evaluation of the IMs, therefore, involves analysis of the dynamic properties of the structure function. Thus, the logic derivatives constitute a perfect tool for evaluation. In this section, we describe how we can use logic derivatives to calculate IMs introduced in section 1.6.

Definition (1.34) of the SI agrees with the relative number of situations (state vectors) in which a failure of a component results in a failure of the system. The definition relates to DPBD (2.10), which describes situations in which a change in the value of a variable results in the same change in the value of the function. Therefore, we can calculate SI in terms of the derivative as [20]:

$$\mathrm{SI}_i = \mathrm{TD}\left(\frac{\partial \phi(1 \rightarrow 0)}{\partial x_i(1 \rightarrow 0)}\right), \tag{2.25}$$

where ϕ is the structure function and $\mathrm{TD}(.)$ denotes the truth density. The truth density is defined as the relative number of elements of the domain of a Boolean-valued function for which the function evaluates to 1. Note that the DPBD is a function of $n - 1$ variables, which we need to consider in the calculation of the relative number of states. Details of the computation of the derivative and the truth density depend on the specific representation of the structure function. We describe the details of the computation in section 3.3.2 and section 3.3.4.

One of the interpretations of the definition of BI is that it agrees with the probability that the failure of a component results in the failure of the system. Therefore, we can compute BI using the DPBD as [20], [41]:

$$\mathrm{BI}_i = \Pr\left\{\frac{\partial\phi(1 \to 0)}{\partial x_i(1 \to 0)} \leftrightarrow 1\right\}. \tag{2.26}$$

Notice that the evaluation of (2.26) involves the calculation of the same DPBD as in (2.25). The difference is in the last step where we calculate probabilities instead of the relative number of states. Again, details of the computation of probabilities depend on the specific representation of the structure function.

As we stated in section 1.6, there exist several generalizations of BI for MSS. Logic derivatives are also useful for the calculation of various generalizations of BI. For example, we can use integrated DPLD (2.21) to compute one of the generalizations of BI for MSS as:

$$\mathrm{BI}_{i,s}^{\gtrless} = \Pr\left\{\frac{\partial\phi\left(h_{\geq j} \to h_{<j}\right)}{\partial x_i(s \to s-1)} \leftrightarrow 1\right\}, \tag{2.27}$$

where $s \in \{1,2,\dots,m_i - 1\}$. The definition and existence of various types of integrated DPLD imply that there exist multiple versions of BI for MSS since the choice of the derivative gives a slightly different meaning to the results, which allows us to pick one that best suits our use case. Also, since the definitions of BI and SI are closely related, for each version of BI we can compute the corresponding SI just by altering the last step of the calculation from the calculation of probabilities to the calculation of the relative number of states. Similarly, we can also calculate the corresponding CI for each version of the BI.

FVI differs from the IMs discussed so far since its definitions (1.40) and (1.41) are based on MCSs. The basic approach to the calculation of FVI involves the enumeration of all MCVs (MPVs). Logic derivatives are also a suitable tool for this task since we can use an extension of the derivative as described in [42]. However, the enumeration of all MCVs (MPVs) is not an efficient solution, especially for systems with a large number of components since the number of MCVs (MPVs) is also very large. Fortunately, a more sophisticated approach exists [43], [44] that also utilizes the extension of the derivative and can calculate FVI without the enumeration of MCVs (MPVs).

The definitions that we presented in this section show that the logic derivatives are indeed a perfect tool for the importance analysis because of the relative simplicity of the definitions and also because we can use one derivative to compute multiple IMs.

2.4 Discrete Function Representation

One of the characteristics of complex systems is that they consist of many components. Therefore, the structure function representing such a system may be difficult to represent. Hence a principal task that must precede the reliability analysis itself is the choice of a suitable representation for the function. The literature provides various representations some of which are suitable for the representation of any discrete function while others are specialized to represent a specific type of discrete function e.g. Boolean function. This section introduces some of the representations that we can apply in reliability analysis.

2.4.1 Arithmetic Expression

The arithmetic expression is one of the simplest representations. We mostly encounter it in the literature since it is easily readable to humans. The expression consists of variables and mathematical operators. We denote variables as x_i where i specifies the index of the variable. The type of mathematical operators depends on the type of function the expression represents. For Boolean functions, the commonly used operators are logical conjunction denoted as $\wedge$ (often omitted from the expression), logical disjunction denoted as $\vee$, and logical negation denoted as $\bar{}$ over an expression. For the MVL functions and integer functions commonly used operators are the min and max operators, which return the minimum and maximum of their arguments respectively.

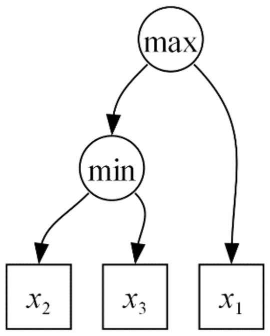


Fig. 2.2 Abstract Syntax Tree representing function $f(\boldsymbol{x}) = \max(x_1, \min(x_2, x_3))$

The main disadvantage of arithmetic expressions is that, although they are easily readable to humans, they are harder to process for a computer. One of the ways to manipulate arithmetic expressions on the computer is to represent them using an Abstract Syntax Tree (AST). AST is a graph structure that consists of nodes representing variables and nodes representing mathematical operators. Fig. 2.2 shows an example of an AST. Though it is easy to evaluate the AST and perform basic arithmetic operations, it is quite complicated to perform more complicated calculations such as the calculation of logic derivatives.

2.4.2 Truth Table

Another very simple discrete function representation is the truth table. The table explicitly assigns the value of the function to each element of the domain of the function. This implies that the size of the table is the same as the size of the domain of the function. Tab. 2.1 shows formulae for the calculation of the size of the table for different discrete function types. For each type, the size depends exponentially on the number of variables n. This property is impractical even for smaller functions of roughly tens of variables. However, thanks to its simplicity, the table is often used in examples and for testing since we can simply (but costly) implement even more complicated calculations such as the logic derivatives.

Tab. 2.2 shows a truth table of an integer function. Besides the straightforward form of the table as shown in Tab. 2.2, there exist techniques that make the table more compact [9]. One of the techniques is to enumerate only elements of the domain in which the function evaluates to a certain value assuming that the function evaluates to the same known value in all other elements. For example, in the case of a Boolean function, we only need to enumerate elements of the domain where the function evaluates to 1 and assume that the function evaluates to 0 in other points. In general, for a function that has a codomain of size m, we need to enumerate $m - 1$ subsets of the truth table.

Tab. 2.1 Size of the domains of different function types where *n* is the number of variables, *m* is the number of values of the MVL function and m_i is size of the domain of ith variable of the integer function

Function type	**Size of the domain**
Boolean	2^n
Multiple-Valued Logic	m^n
Integer	$m_1 * m_2 * \dots * m_n$

Tab. 2.2 Truth table of the integer function $f(\boldsymbol{x}) = \max(\min(x_1, x_2), x_3)$ where $x_1, x_2 \in \{0,1\}$ and $x_3 \in \{0,1,2\}$

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{x_3}$	$\boldsymbol{f}$	$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{x_3}$	$\boldsymbol{f}$
0	0	0	0	1	0	0	0
0	0	1	1	1	0	1	1
0	0	2	2	1	0	2	2
0	1	0	0	1	1	0	1
0	1	1	1	1	1	1	1
0	1	2	2	1	1	2	2

2.4.3 Truth Vector

A truth table like the one shown in Tab. 2.2 has a regular structure. We can use this property to make the table more efficient by storing only the last column of the table. We call the column a *truth vector*. An important property of the vector is that we can access its elements using an index l. Therefore, we need to map the elements of the domain of the function to the index l. For an integer function the index can be calculated as follows:

$$\begin{aligned} l &= \sum_{i=1}^{n} o_i x_i \text{ where} \\ o_n &= 1 \\ o_i &= m_{i+1} o_{i+1} \text{ for } i = 1,2,\dots,n-1. \end{aligned} \tag{2.28}$$

The auxiliary vector o_i is called an information vector. It is beneficial to calculate the information vector only once before any further calculations. Furthermore, in the case of the MVL function and Boolean function, the calculation of the information vector simplifies since the formula (2.28) simplifies to (2.29) and (2.30) respectively.

$$\mathrm{l} = \sum_{i=1}^{n} m^{n-i} x_i, \tag{2.29}$$

$$\mathrm{l} = \sum_{i=1}^{n} 2^{n-i} x_i. \tag{2.30}$$

The size of the vector is equal to the number of elements of the domain of the function. Fortunately, we can utilize the way computers store numbers to make vectors more compact. The smallest addressable unit of memory is a byte that can encode 256 unique values. However, the sizes of domains of integer variables are usually much smaller, and in the case of Boolean variables it is just two values i.e., we can encode it using a single bit. Therefore, we can store the truth vector more efficiently if we encode multiple elements of the domain using a single byte. Let p be the maximum of the sizes of domains of variables of an integer function and let n be the number of variables. Then one byte can store $r = \lfloor \log_2 p \rfloor$ elements of the domain and therefore the size of the truth vector is reduced by the factor of r. Though this optimization is useful, especially for Boolean functions, it does not help with the eventual exponential complexity. For example, the memory requirements of a truth vector representing a Boolean function of 40 variables using the described optimization are 128 GiB of memory. The truth vector becomes large even for dozens of variables. Nonetheless, the truth vector is useful for testing as a representation of the truth table.

2.4.4 Decision Tree

An important property of the truth table is that we can look up the value of the function corresponding to a given element of the domain. The truth vector allows the lookup using a simple computation. Another useful technique is to use a so-called Decision Tree (DT). The decision tree is a specific type of graph that satisfies the tree invariants [45]. A special kind of DT is a Binary Decision Tree (BDT) that represents a Boolean function. We can use the decision tree to look up the value of a function using a series of decisions.

A decision tree (Fig. 2.3) consists of two types of nodes. The first type is *internal nodes* that represent variables and the second type is *terminal nodes* that represent values from the domain of the function. Each internal node is associated with one variable x_i and has a tuple of m_i outgoing edges. The k^{th} edge in the tuple represents a situation in which the variable x_i has the value k. To look up a value of the function we start at the root of the tree. In each internal node, we choose an edge depending on the value of the variable. Using the edge, we move to the next node. We repeat this process until we reach a terminal node. In Fig. 2.3 the bold edges represent the path in the tree for values of variables $x_1 = 1, x_2 = 0, x_3 = 2$, which we can shortly denote using the vector notation $(1,0,2)$.

The decision tree is that it is ordered – each level of the decision tree either contains internal nodes associated with the same variable or contains only terminal nodes (last level). It follows that the size of the tree is exponential in the number of variables n and therefore the tree is impractical for the representation of bigger functions – just like the truth table. The graph approach is a basis for a more sophisticated data structure – the decision diagram.

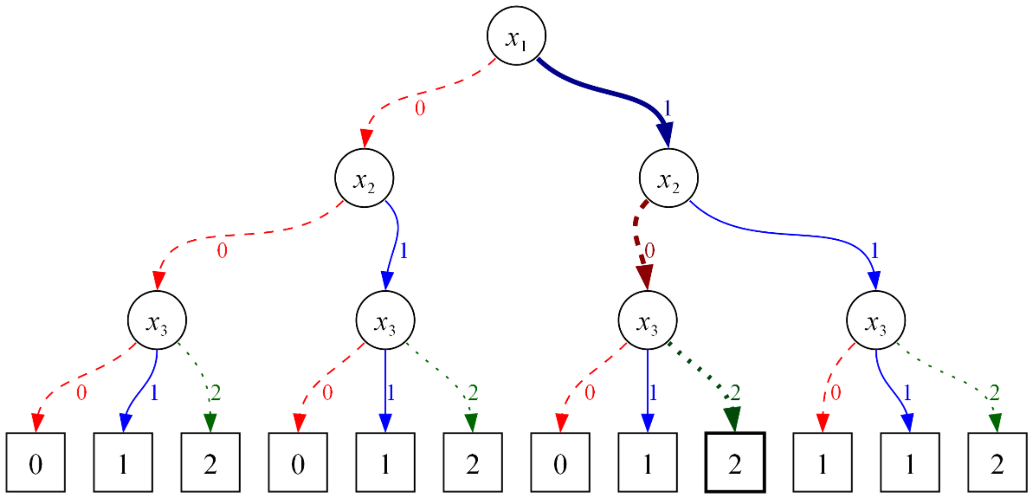


Fig. 2.3 Decision tree representing the integer function defined in Tab. 2.2

2.4.5 Decision Diagram

The decision diagram builds on the idea of the decision tree – to represent a discrete function using a graph structure. It enhances the graph structure (which no longer is a tree) to allow for a more compact representation of a discrete function. Like the decision tree, a decision diagram consists of internal and terminal nodes serving the same purpose as in the decision tree. The nodes may or may not be ordered depending on a specific type of decision diagram. Fig. 2.4 shows an example of a decision diagram representing the same function as DT in Fig. 2.3.

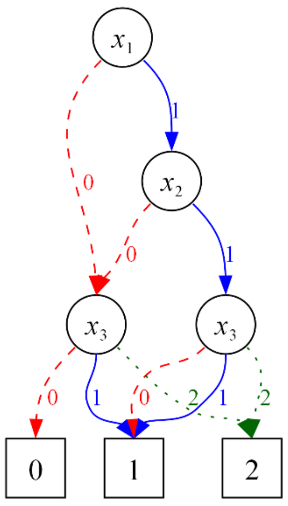


Fig. 2.4 Decision diagram representing the same function as DT in Fig. 2.3

Decision diagrams are the central topic of this thesis, thus, the rest of the chapters provide an in-depth description of the fundamental properties of decision diagrams and their application in reliability analysis.

3 Decision Diagrams

A decision diagram is a graph structure that can efficiently represent discrete functions. Researchers have proposed various types of decision diagrams over time. Some of the diagrams are intended for general discrete function manipulation whilst others aim to solve a specific problem. This chapter introduces several types of decision diagrams along with their key properties and algorithms for their creation and manipulation. Finally, the chapter describes the application of decision diagrams in reliability analysis as well as diagram algorithms specific to reliability analysis.

3.1 Reduced Ordered Decision Diagrams

The Binary Decision Diagram (BDD) proposed by Lee [46] and further developed by Akers [47] and Bryant [11] as Reduced Ordered Binary Decision Diagram (ROBDD) is, historically, the first decision diagram. The literature often refers to ROBDD as just a Binary Decision Diagram (BDD) since this version is the most widely used. In this thesis, we will also use the term BDD instead of ROBDD for brevity. BDD is a graph structure designed for the representation of Boolean functions However, theoretical principles and techniques used in its definition are the basis for most of the other diagrams described in this chapter.

As we described in Section 2.1 the Boolean function (2.2) is a special type of discrete function (2.1). Therefore, it is natural that techniques and approaches utilized in its representation in the form of the BDD were considered and used in the development of similar techniques for the representation of other types of discrete functions – namely the MVL function (2.3) and integer function (2.4). The authors proposed the Reduced Ordered Multi-valued Decision Diagram (ROMDD) [12] as a generalization of the (RO)BDD to represent these functions. Like (RO)BDD, the literature often refers to ROMDD shortly as MDD. We will also use this notation in this thesis.

Though historically, decision diagrams have been developed from simpler BDD to more general MDD and others, we will proceed with the description of MDD in its most general form which represents an integer function. This approach is more concise since BDD and MDD representing MVL function are only a special case and, hence, do not require separate descriptions. In Fig. 3.1 we can see all three diagram types mentioned – BDD as the simplest type and two versions of MDD, the first one representing an MVL function and the second one representing an integer function.

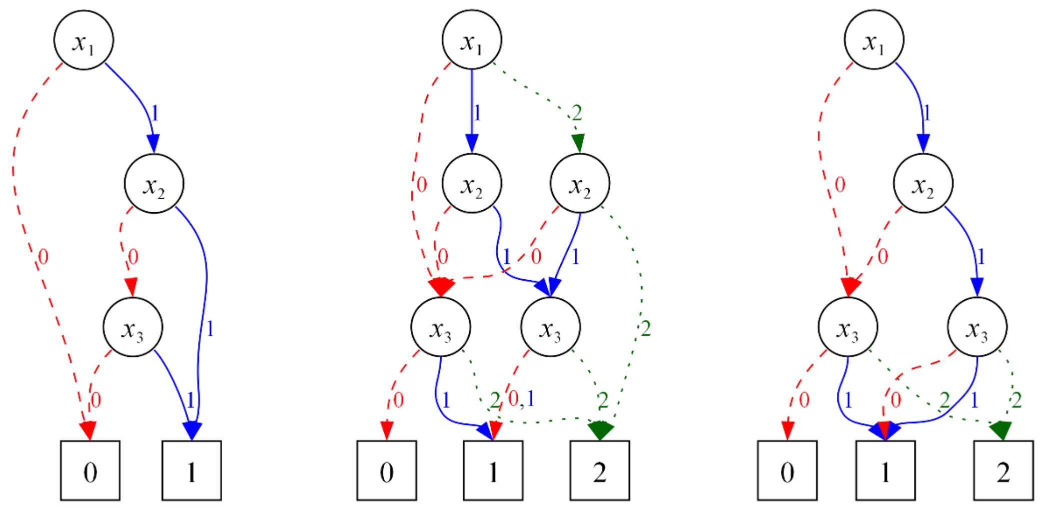


Fig. 3.1 Left: BDD representing Boolean function, middle: MDD representing MVL function, right: MDD representing integer function

3.1.1 Graph Structure

MDD is a graph structure that consists of *terminal nodes* that represent values of the function and *internal nodes* that represent variables. A terminal node is identified by the value it represents. Let us denote nodes using capital letters $A, B, \dots$. Then, we denote the value of a terminal node A as $\textsc{Value}(A)$. Also, we use the notation T_a to denote a terminal node representing value a. In Fig. 3.1 and all other figures, we denote terminal nodes using a square shape with a number representing the value. An internal node is associated with a variable x_i and a tuple of m_i edges leading to other nodes – *sons*[1] of the node. The edges represent possible values of the variable – k^{th} edge represents value k for $k = 0,1, \dots, m_i - 1$. Let B be an internal node. Then, we denote the index i of the variable it is associated with as $\textsc{Index}(B)$ or shortly as i_B and its k^{th} son as $\textsc{Son}(B, k)$ or shortly as B_k.

One of the principal operations that we can perform on MDD is to evaluate it for some specific input vector $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$. During the evaluation, we repeat a decision in each internal node (starting at the root node) such that we choose an edge according to the value of x_i. The edge leads to another node, which is either an internal node – in which case we repeat the process, or it is a terminal node that contains the value of the function corresponding to $\boldsymbol{x}$. An alternating sequence of internal nodes and edges that lead to a terminal node is known as a *path* [48].

[1] The term son or direct successor is typically used with tree structures. However, it is short, descriptive and conveys the intended information very well also in the case of decision diagrams.

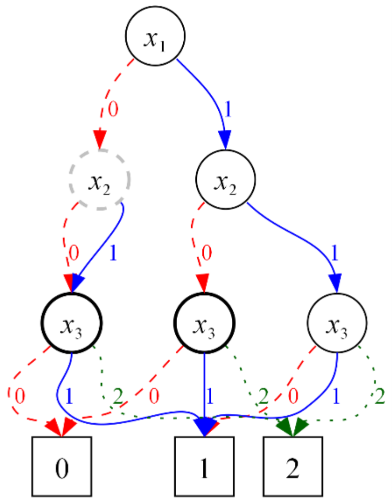


Fig. 3.2 MDD containing redundant node (dashed gray outline) and two duplicate nodes (thick black outline)

Nodes of an MDD are ordered on levels. The last level contains terminal nodes. Other levels contain internal nodes and all internal nodes on the same level are associated with the same variable. We denote the level of node A as $\textsc{Level}(A)$. The first level contains a single node – the root node – denoted as $root$. Consequently, using this notation, we may write the following relations $\textsc{Level}(root) = 1$ and $\textsc{Level}(T_a) = n + 1$ for $a = \{0,1, \dots, m - 1\}$.

Edges in MDD can only lead to nodes on lower levels. This rule ensures that nodes are in the same order on each path from the root node to a terminal node. This property is captured in the Reduced *Ordered* MDD part of the full name. Consequently, the order of variables is one of the properties that we can describe for each ordered MDD. In Fig. 3.1 (and in most of the other figures depicting diagrams) we use the so-called implicit order, which orders variables by their indices i.e., x_1 on the first level, x_2 on the second level all the way to the x_n on the second to last level i.e., it holds that $\textsc{Index}(A) = \textsc{Level}(A)$.

Another important property of MDD is that it does not contain any *redundant* nodes and no *duplicate* nodes. The two properties ensure that each node in the diagram is *unique*. A redundant node is a node with all edges leading to the same node. Decisions in such a node will always result in the selection of the same son (e.g. during the evaluation). Therefore, there is no need to keep such a node in the diagram. In Fig. 3.2 we can see an example of a redundant node (representing variable x_2) marked with a dashed gray outline. Two nodes are duplicates if they are roots of two isomorphic subgraphs. Therefore, only one node from the group of duplicate nodes is always retained. This property is captured in the *Reduced* Ordered MDD part of the full name. More examples of redundant and duplicate nodes can be found in section 3.2.2.

3.1.2 Mathematical Foundations

The mathematical foundation of decision diagrams lies in a recursive application of Shannon's expansion [49] (for BDD) and generalized Shannon's expansion [9] (for MDD). The definition of Shannon's expansion uses the cofactor of a discrete function (2.6). The Shannon's expansion with respect to variable x_i is defined in terms of the cofactor as:

$$f(\boldsymbol{x}) = x_i f(1_i, \boldsymbol{x}) \vee \overline{x_i} f(0_i, \boldsymbol{x}), \tag{3.1}$$

for the Boolean function and the generalized version for an integer function is defined as:

$$f(\boldsymbol{x}) = \sum_{k=0}^{m_i - 1} \left(\{x_i \leftrightarrow k\} * f(k_i, \boldsymbol{x}) \right), \tag{3.2}$$

where $i \in \{1,2, \dots, n\}, k \in \{0,1, \dots, m_i - 1\}$ and $\leftrightarrow$ represents the logical biconditional (1.10). The expansion (3.1) splits the function into two cofactors of the function – $f(1_i, \boldsymbol{x})$ and $f(0_i, \boldsymbol{x})$. The disjunction of two conjunctions selects exactly one of the cofactors based on the value of the variable x_i. The selection or more suitably – the *decision*, is neatly represented by an internal node, which can be considered a graph representation of Shannon's expansion with respect to variable x_i. The generalized Shannon's expansion (3.2) follows the same rationale by selecting exactly one of the m_i cofactors of the function with respect to variable x_i. The decision is achieved by the logical biconditional $\{x_i \leftrightarrow k\}$ evaluating to 1 for exactly one value of k and to 0 for all other values. The multiplication then selects only one of the cofactors. Fig. 3.3 illustrates the relationship between the expansions (3.1) and (3.2) and an internal node of a decision diagram.

Fig. 3.3 shows that each node except the root node on a certain level of the diagram represents a cofactor of some function from the above levels. The expansion (3.2) effectively splits the domain of the function into m_i parts of the form $(a_i, \boldsymbol{x})$ for $a = 0,1, \dots, m_i - 1$. In each part, the value of the variable x_i is known and the number of variables is reduced by one. Therefore, the cofactors (son nodes) are either constant functions (terminal nodes) or they are internal nodes representing another recursive expansion. The recursion is guaranteed to terminate after at most n expansions since values of all variables are necessarily known at that point. This sets the upper bound on the number of levels of a reduced ordered decision diagram, which is n. However, the cofactor can turn to a constant function sooner when it evaluates to the same value for all elements of its domain. This agrees with the situation when an edge in a diagram skips some levels and goes directly to a terminal node.

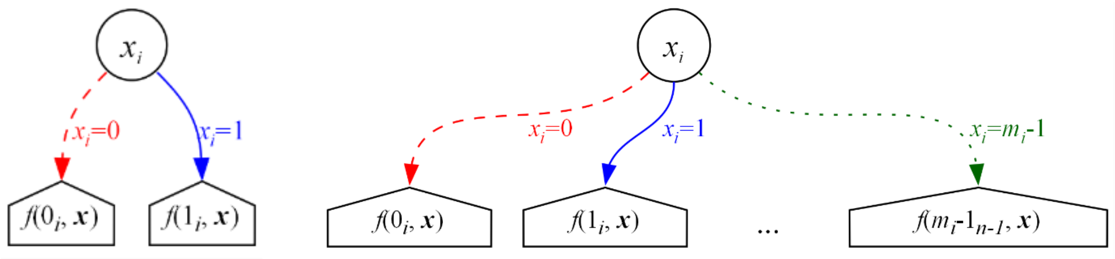


Fig. 3.3 Internal node of a BDD (left) and MDD (right) that represents Shannon’s expansion with respect to i^{th} variable

Lastly, Fig. 3.3 shows an important property of the decision diagrams that not only the root, but each node represents a discrete function (even a terminal node, which represents a constant function). Therefore, when suitable, we can use the term function and node interchangeably.

3.1.3 Canonical Representation

The *reduced* and *ordered* properties of MDD ensured that MDD is a *canonical representation* of a discrete function. This was proven by Bryant [11] for BDD and later by the authors in [12] for MDD. Canonical representation ensures that each function has a unique representation. An example of a representation that is not canonical is an arbitrary expression. For instance, Boolean functions f_1 and f_2 defined by expressions $f_1(\boldsymbol{x}) = x_1x_2 \vee x_2x_3$ and $f(\boldsymbol{x}) = \overline{x_1}x_2x_3 \vee x_1x_2\overline{x_3} \vee x_1x_2x_3$ represent the same function even though they contain different numbers of terms and different operators. In general, to check whether two non-canonical representations of discrete functions represent the same function, we transform each representation into some other – canonical – representation and compare them for equality.

In the case of MDDs, we need to define the equality of two nodes to be able to compare two diagrams. The first condition for nodes A and B to be equal is that they must both be either terminal nodes or internal nodes. If they are terminal, they are equal if and only if $\text{Value}(A) = \text{Value}(B)$. If they are internal, they are equal if and only if $\text{Index}(A) = \text{Index}(B)$ and $A_k = B_k$ for $k = 0,1,\dots,m_{i_A} - 1$. Notice the recursive nature of equality comparison, which shows that each node can be considered as a root of (sub)diagram on its own i.e., each node (not only the root node but even terminal nodes) represents a unique function. Implementation of the comparison using the definition would be relatively complicated since it would require simultaneous traversal of both diagrams. Hence, in section 3.2.1 we describe an efficient approach that is used in practice.

Another typical example of a canonical representation is the truth table which. As we described in section 2.4.2 and section 2.4.3, the truth table is quite inefficient for the representation of larger functions. However, it is useful for testing and verification because of its simplicity. Lastly, let us note that even though expressions, in general, are not canonical representations, some expressions with certain restrictions such as Disjunctive Normal Form [8] are canonical representations.

3.1.4 Number of Internal Nodes

One of the key properties of each discrete function representation is its size, specifically, the relation of the size to the number of variables n. The reason is the practical limitation posed by the limited amount of memory and complexity of the algorithms that work with the representation, which usually depends on the size. When we described other discrete function representations such as the truth table or the decision tree, we also provided a formula to calculate the size of the representation for a given number of variables n (see Tab. 2.1, and section 2.4.4). Unfortunately, it is not possible to do the same calculation for decision diagrams in general. Nevertheless, we can calculate the worst possible size of a diagram. Consider MDD representing m-valued logic function of n variables. Then we can calculate the upper bound on the number of nodes as [50]:

$$\min_{h}\left(\frac{m^{n-h}-1}{m-1}+m^{m^{h}}-m\right), \tag{3.3}$$

where $0 \leq h \leq n$. To get a better insight into the reasoning behind the expression we need to follow constraints of the structure of a decision diagram [51]. There is only one node on the first level of the diagram. At the second level, there can be at most m nodes (sons of the root node). For the next levels, the number of possible unique nodes grows exponentially following the branching of a decision tree. However, at the same time, there can be at most m terminal nodes at the last level of the diagram. Since each node in the diagram is unique and edges can only lead to nodes at lower levels the number of nodes at the second to last level (last internal level) is also limited. We can see that the size of the diagram grows exponentially from the top and combinatorically from the bottom. The increasing sequences “meet” at some internal level given by h, which minimizes the expression (3.3). Therefore, the first term of the addition accounts for the exponential growth of the diagram from top to bottom, and the second term accounts for the combinatorial growth from the bottom of the diagram.

Tab. 3.1 Number of nodes in BDDs representing specific (symmetric) functions

Function	**Number of internal nodes**
Logical conjunction/disjunction	n
Odd parity function	$2n - 1$
Structure function of a k-out-of-n system	$k(n - k + 1)$

The exponential upper bound on the number of nodes does not seem to improve the exponential complexity of the decision tree and the truth table. However, for many functions that we encounter in practice, the size of the diagram is much more favorable. A typical example is the representation of symmetric functions that we briefly describe in section 3.1.5. In Tab. 3.1 we can see formulas to calculate the number of internal nodes in BDDs representing such functions (the number of terminal nodes is always two, except in the special case of a constant function). Notice that the numbers are much better than exponential. Also, the size of the diagram is closely tied to the order of variables, which we discuss in section 3.1.5.

3.1.5 Order of Variables

The order of variables in a decision diagram is one of the properties required for it to be a canonical representation. A well-known consequence of this property is that it might influence the size – the number of nodes – of the diagram [11]. We can classify diagrams into two groups. The first group contains diagrams whose size does not depend on the order of variables and, naturally, the second – larger group – contains diagrams whose size depends on the order of variables.

Diagrams in the first group represent *symmetric functions* – functions that evaluate to the same value regardless of the order of their arguments. Diagrams representing such functions have a regular structure that does not change for different orders of variables. Typical examples of such functions are the Boolean parity function, logical conjunction of n variables, logical disjunction of n variables, min function, max function, etc. Fig. 3.4 shows BDDs representing the Boolean parity function of three variables. BDDs in the figure have different orders of variables and they have the same regular ladder-like structure.

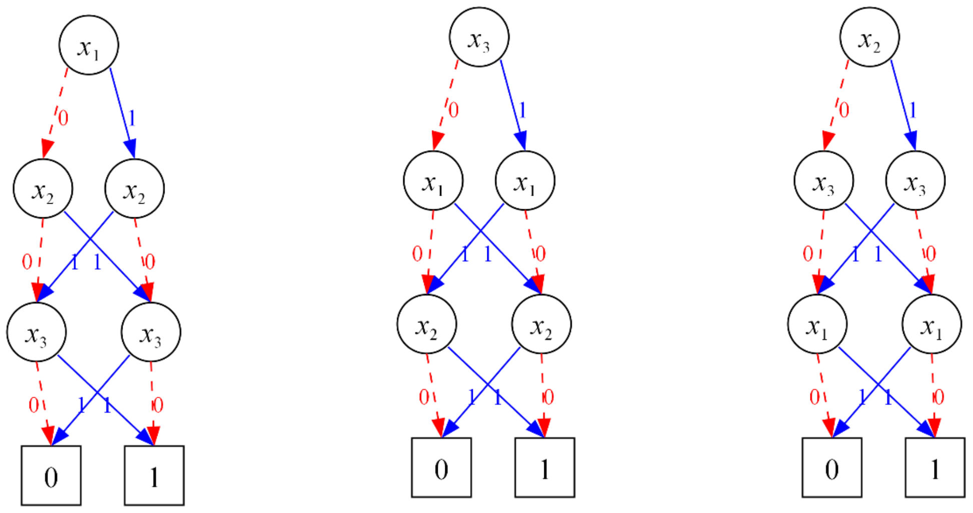


Fig. 3.4 BDDs with three different orders of variables representing parity function

The second group of diagrams is more important since we encounter them more often and therefore are important to consider in the design of software tools for the creation and manipulation of decision diagrams. The order of variables can be a factor that decides whether we can construct a diagram within reasonable time and memory constraints. For example, Fig. 3.5 shows BDDs representing Boolean function $f(\boldsymbol{x}) = x_1x_2 \vee x_3x_4 \vee x_5x_6$ with two different orders of variables. Notice that even for such a simple function the number of nodes grows significantly. Therefore, it is vital to use a suitable order of variables. Unfortunately, finding an optimal order is an NP-complete problem [52]. Hence, in practice, we are reliant on the use of a variety of heuristic approaches.

The literature describes two principal types of heuristics for the choice of the order of variables [53], [54], [55]. The first type is *static heuristics*. Their key property is that they choose the order of variables before the creation of any diagrams. Then, during and after the creation the order stays the same. The advantage of the static approach is that it can exploit the properties of the specific problem and therefore find a better order for that specific problem. On the other hand, it might not be possible to use the approach for generic diagram manipulation tools.

The main idea behind the heuristics of the second type is to gradually re-order variables as the diagrams are created, hence, those heuristics are called *dynamic heuristics*. Since they are opposite to the static ones their advantage is that they do not need any knowledge about the function(s) the diagram(s) represent. This allows them to be used in generic decision diagram manipulation tools. Presumably, the most well-known dynamic heuristic is variable sifting [55] originally proposed for BDDs.

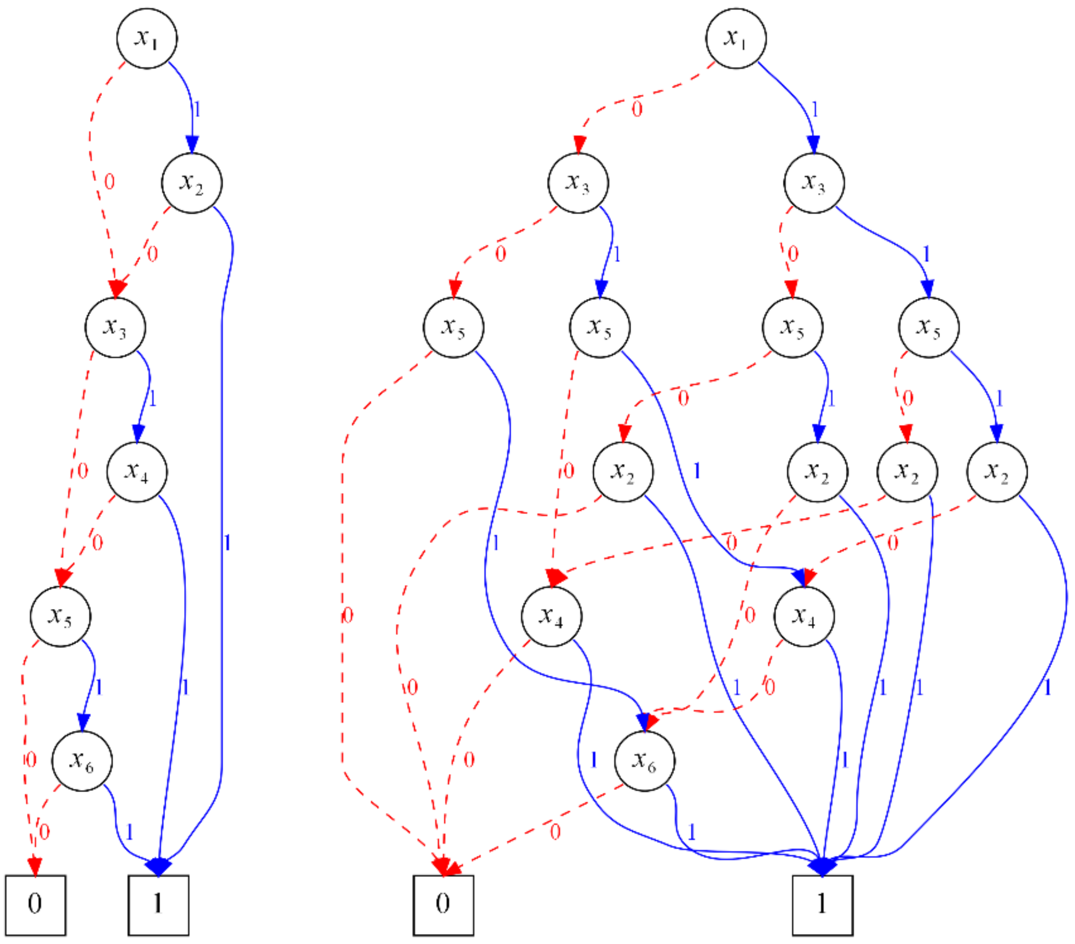


Fig. 3.5 BDDs representing the same function using a different order of variables

A vital operation for each dynamic heuristic is a swap of indices between two adjacent levels in a diagram so it can gradually adjust the order of variables using a series of swaps. The swap operation needs to maintain all invariants of the diagram – mainly that each node represents a unique function, and the function the node represents does not change during the lifetime of the node. The implementation of the swap operation is based on the observation that to swap nodes that represent variable x_{i_1} with nodes that represent variable x_{i_2} on the next level, we only need to examine nodes at two levels and modify nodes representing variable x_{i_1} [56], [57].

To swap a node we modify it by changing its index from i_1 to i_2 and set its sons to new nodes with index i_1 and sons that were grandsons of the original node in such a way, that the function is preserved at the node. Fig. 3.6 shows the swap operation for a specific example. Bold darker edges show part of a path in the diagram. The important thing to notice is that the path (and all other paths) leads to the same grandson after the swap. After the swap, the old sons of the node can be freed (if they are not shared by some other nodes in the diagram), also the newly created nodes might be redundant, so they are not created. This is the point where a series of swaps can result in a new diagram that contains fewer nodes.

An interesting observation is that the swap operation practically mutates the state of the node by changing pointers to its sons and its index, however, logically the node represents the same function and therefore it does not violate the immutability invariant.

The variable sifting heuristic described in [55] uses a series of swaps to find a better order of variables. It places each variable on a level where the total number of nodes in the diagram is the smallest. It does so by first trying to place a variable (by swapping all nodes associated with a given variable) on each level of the diagram and subsequently restoring the best-observed case. The order in which variables are placed is given by the total number of nodes initially associated with the given variable starting with the variable with the highest number of nodes.

Although heuristics can help a lot in the practice there exist some functions for which the number of nodes in the diagram will depend exponentially on the number of variables regardless of the order of variables. We call such functions inherently complex functions [11]. An example of an inherently complex function is a Boolean function describing an integer multiplier [11].

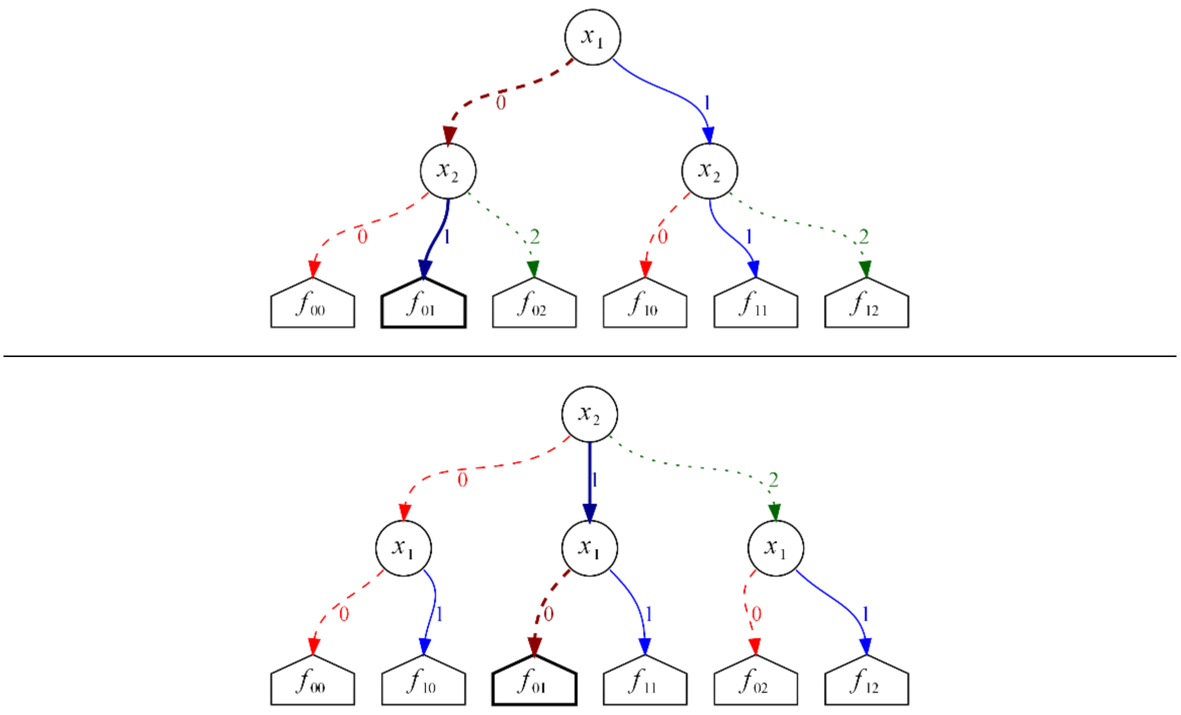


Fig. 3.6 Node of an MDD before (top) and after (bottom) the swap

3.1.6 BDD Extensions and Alternatives

Since its introduction, BDD has proven to be a fundamental tool for solving problems in areas such as logic synthesis [58] or formal verification [59]. The reason for its popularity among researchers is that it can represent Boolean functions efficiently and is supported by various software libraries (see section 3.2).

Naturally, a general tool – despite being reasonably efficient – cannot exploit the specifics of a given problem. Therefore, the researchers have proposed various modified versions of the BDD structure that are designed to solve a narrower set of problems. BDD with complemented edges [60], [61] simplifies the representation of complemented Boolean functions. Zero-suppressed Decision Diagram (ZDD) [62] improves diagram sizes for Boolean functions representing sets (especially sparse sets). Algebraic Decision Diagram (ADD) [63] allows more terminal nodes than just two nodes representing values 0 and 1. And Edge-Valued Binary Decision Diagram (EVBDD) [64] also allows more terminal nodes than just two nodes representing values 0 and 1.

In addition to the above-named BDD extensions, researchers have proposed several other modifications – using various forms of binary encoding of multi-state variables and functions – such as logarithmic BDDs (LBDD) [65], Multi-State BDDs (MBDD) [66], or Multi-Rooted Binary Decision Diagrams [67]. Finally, in Fig. 3.7, we can see examples of selected BDD extensions.

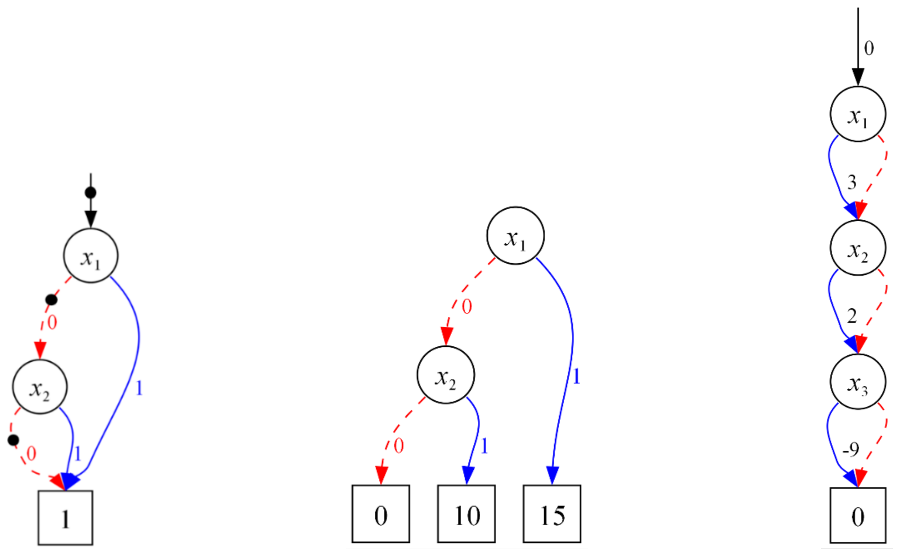


Fig. 3.7 BDD with complemented edges representing the function $f_1(\boldsymbol{x}) = \overline{x_1 \vee \overline{x_2}}$ (left); ADD representing function $f_2(\boldsymbol{x}) = \max(15x_1, 10x_2)$ (middle); EVBDD representing function $f_3(\boldsymbol{x}) = 3x_1 + 2x_2 - 9x_3$ (right)

3.2 Decision Diagram Implementation

Researchers have developed several software libraries implementing decision diagrams in different programming languages. The libraries are often referred to as decision diagram packages. The most well-known are the BuDDy [68] and CUDD [69] written in C language with an interface for C++ and the more recent Sylvan [70] parallel BDD package written in C language. Several programming languages offer libraries that serve as interfaces to call these libraries such as CUDD for Haskell [71] or dd for Python [72]. Furthermore, several implementations in other programming languages exist, such as the JDD [73] library for Java and the DecisionDiagrams library [74] for C#. Tab. 3.2 contains an overview of the libraries.

As we can see in the table, most implementations support only BDDs and some of their alternatives. We aim to examine and develop techniques for the analysis of MSS using MDDs – the examination requires a performant software library supporting MDDs. Since none of the state-of-the-art C libraries support MDDs, we implemented our open-source decision diagram library called TeDDy – **T**emplated **De**cision **D**iagram librar**y** [75], [76] in the C++ language. The goal of the library is to provide general tools for the creation and manipulation of BDDs and MDDs with a module dedicated to reliability analysis that utilizes decision diagrams.

Tab. 3.2 Overview of selected decision diagram packages

Package	**Language**	**Supported diagrams**
BuDDy	C, C++	BDD
CUDD	C, C++	ADD, BDD, ZDD
Sylvan	C, C++	ADD, BDD, ZDD
CUDD (Haskell)	Haskell	ADD, BDD, ZDD
dd	Python	BDD, MDD
JDD	Java	BDD, ZDD
DecisionDiagrams	C#	BDD

As the name suggests, the library uses the powerful template mechanism of the C++ language to implement core functionalities universally using object-oriented programming while maintaining runtime performance comparable to state-of-the-art C libraries. Implementation of the library uses the following layers:

- node representation,

- node management,
- diagram management,
- and user-facing interface.

The four-level design of the library can be seen in Fig. 3.8. It allows higher levels to reuse the core low-level parts of the library – e.g. the management of nodes.

Section 3.1 covers the theoretical aspects of reduced ordered decision diagrams – describing their characteristic properties and mathematical foundation. However, a diagram implementation that would just blindly follow the above definitions would not be effective. Therefore, in the rest of this section, we focus on important aspects of the implementation of software tools for the creation and manipulation of decision diagrams – focusing on the implementation of MDDs representing integer functions. A complete implementation of a decision diagram package needs to address the following problems:

- representation of graph nodes;
- management of graph nodes – node sharing;
- diagram creation – static, dynamic, and direct approaches;
- diagram transformations;
- examination of diagram properties – efficient diagram traversals and memoization techniques.

Our software library TeDDy implements all the above-mentioned techniques with a focus on performance and extensibility to allow the implementation of reliability analysis algorithms on top of the decision diagrams.

In the following chapters and sections of this thesis, we utilize pseudocodes to better illustrate algorithms and their properties. Some of the pseudocodes describe existing algorithms while others describe new algorithms, which are contributions of this thesis. Therefore, to differentiate between the two cases, the existing algorithms are enclosed in appendix A, while novel algorithms are presented directly in the main sections. Finally, considering the implementation aspects, the pseudocodes assume, for simplicity, that the diagrams use the default order of variables i.e., that for an internal node A it holds that $\textsc{Index}(A) = \textsc{Level}(A)$.

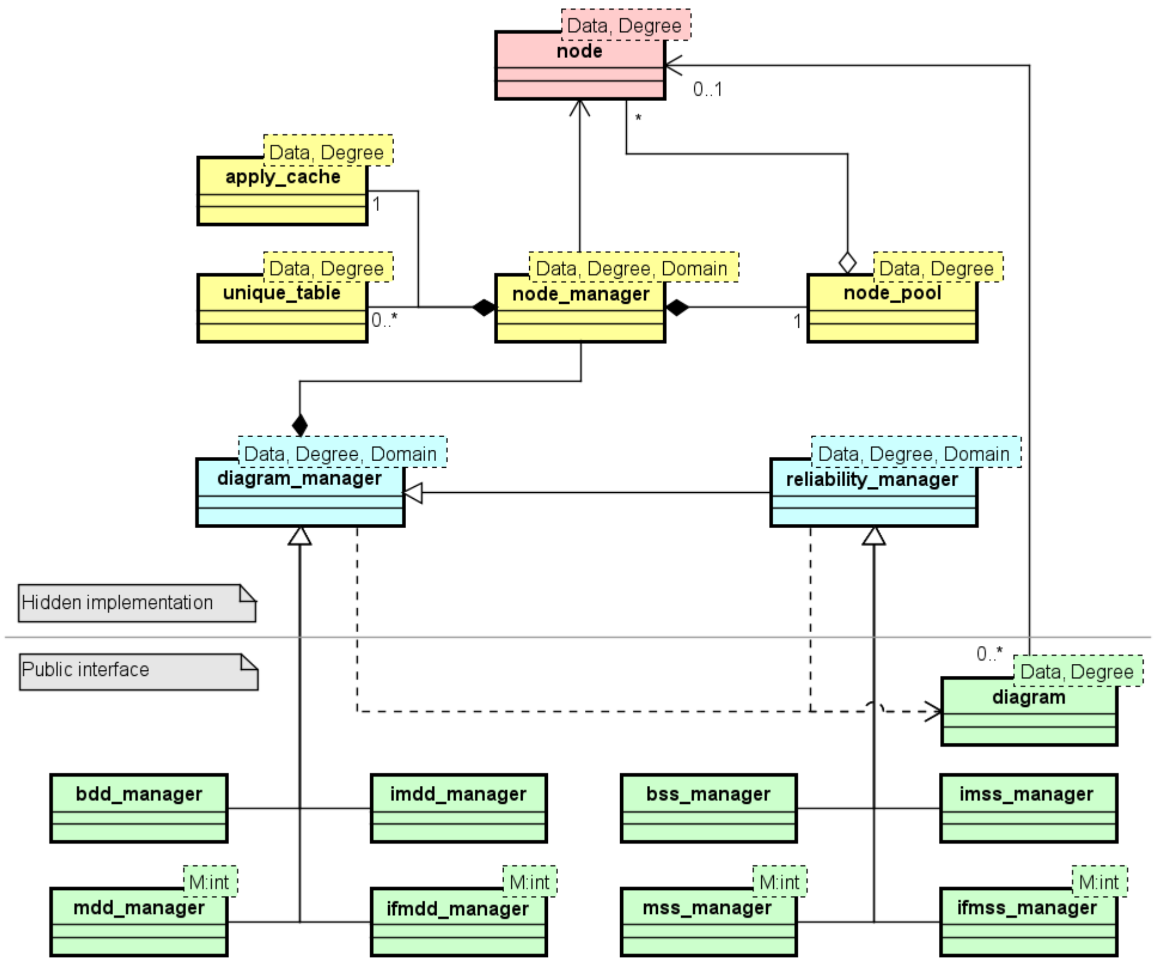


Fig. 3.8 UML class diagram showing the most important classes on four layers of the TeDDy library

3.2.1 Node Sharing

The sharing of isomorphic subgraphs within a single diagram is one of its essential properties. It greatly contributes to the efficiency of the structure and is also one of the reasons that the diagram is a canonical representation of a function.

Each node in the diagram is unique and thus represents a unique function. The function represented by a node does not change during the lifetime of the node. The diagram as a data structure can therefore be regarded as a persistent data structure. However, let us recall that the actual bytes representing a node may change e.g., during a swap of variables (section 3.1.5) but the logical meaning of the node (the function it represents) stays the same. This invariant property allows for more optimizations that we describe further in this section.

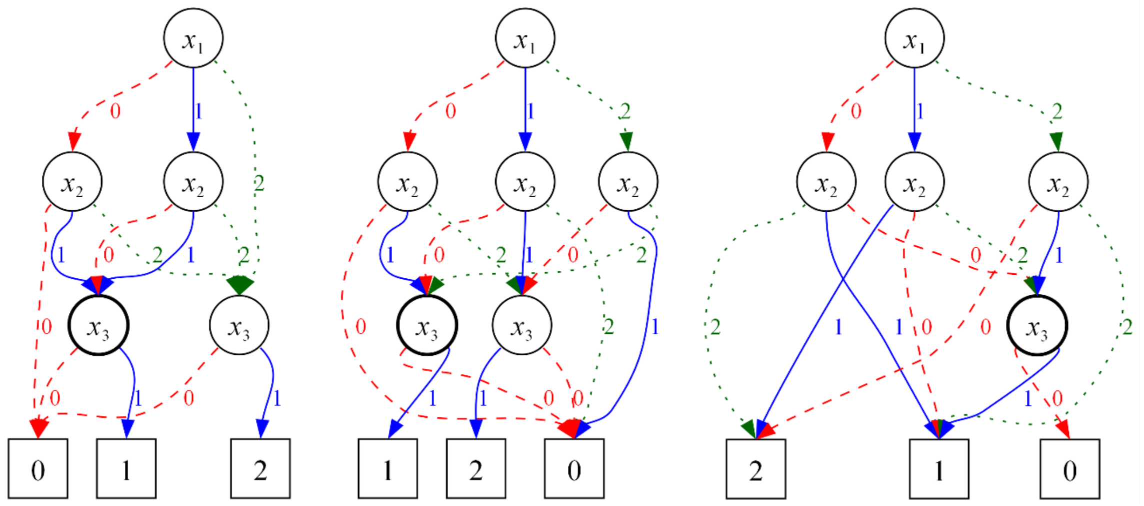


Fig. 3.9 Three MDDs represent integer functions each containing node representing the same function

Further improvement of the diagram structure lies in the development of the idea of node sharing beyond a single diagram. The reason for this is that a node representing a certain function usually appears in multiple diagrams created separately and thanks to immutability we are certain that it will always represent the same function. For example, in Fig. 3.9 we can see that the node marked with the bold outline, representing function $f(\boldsymbol{x}) = x_3$, is part of all three diagrams. Since this is true for other nodes as well, the potential for improvement of the diagram structure is considerable. Instead of maintaining each node unique within a single diagram, we manage a graph in which unique nodes are shared across multiple diagrams. Such a graph has multiple roots (nodes that do not have any incoming edges), therefore, it is sometimes referred to as a multi-rooted directed acyclic graph in the literature. Diagrams represented using this technique are called Shared Decision Diagrams – first proposed for BDDs [77] and later generalized for MDDs [78]. Fig. 3.10 shows the same diagrams as Fig. 3.9 but represented as shared diagrams – using a single graph. Notice that the above-mentioned node representing function $f(\boldsymbol{x}) = x_3$ is present only once.

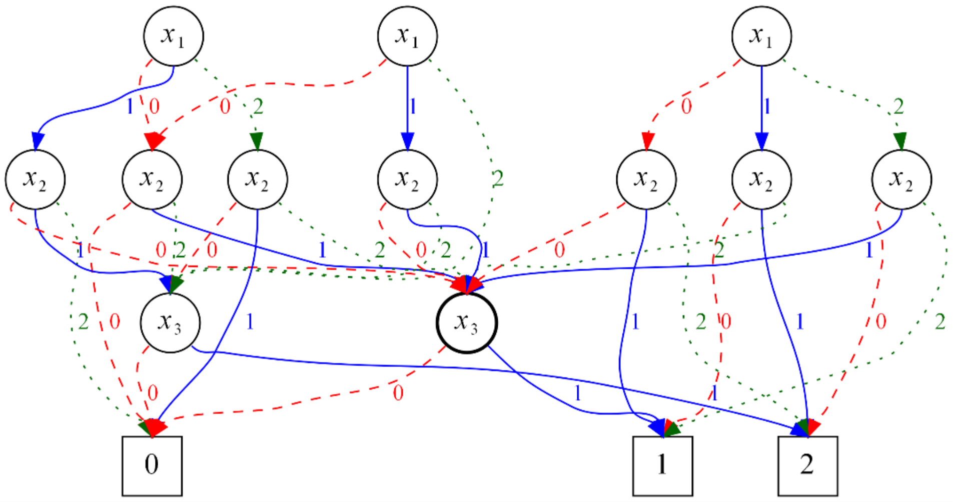


Fig. 3.10 Decision diagrams from Fig. 3.9 are represented with a single multi-rooted graph

To maintain the uniqueness of internal nodes, we use a lookup table called *unique table*. The key for this table is a pair $\left(i, \left(A_0, A_1, \dots, A_{m_i}\right)\right)$ and the value stored in the table is a pointer to the node A. For the terminal nodes, we use a similar table in which the key is the value represented by the terminal node. Such tables are an essential component of every decision diagram library. It is crucial to avoid the direct creation of new nodes. Instead, it is necessary to use dedicated factory functions that work with the unique tables. In the following description, we will refer to the functions as CREATETERMINALNODE (Alg. A.1) and CREATEINTERNALNODE (Alg. A.2). If we exclusively use the functions to obtain new nodes, it cannot happen that two isomorphic subgraphs exist in the graph. Let us notice that this approach is an implementation of the *Flyweight* design pattern [79].

Section 3.1.3 described MDD as a canonical representation. Such property is closely tied with a comparison for equality. Comparison following the definition described in section 3.1.3 involves traversing both diagrams simultaneously and comparing their structure in the process. Computational complexity of such a process is $O(\mathscr{s})$ where $\mathscr{s}$ is the number of nodes in the smaller diagram – in the worst case, we traverse the entire smaller diagram and find out that diagrams are equal/not equal in the last traversed node. A representation that requires such exhaustive comparison is known as a *weak* canonical form [61]. If nodes are not shared between different diagrams, then diagrams representing the same function may lie in a different location in the memory. Therefore, we need to exhaustively compare their structure to check whether they are the same. However, in a graph of shared diagrams that only contains unique nodes, it is sufficient to compare only the pointers (identities) of the root nodes since the same functions are necessarily represented by the same node. Such a representation is known as a *strong* canonical form [61].

3.2.2 Diagram Creation

The unique table and factory functions provide the foundation for the creation of arbitrary MDD. MDDs can be created using different approaches. In this section, we describe the main rationale behind each approach, its advantages, and disadvantages. Finally, we show how some of the approaches can be combined and thus made more efficient.

3.2.2.1 Static Creation

Decision Tree (DT) that we described in section 2.4.4 is a graph structure made of the same type of nodes as MDD. However, unlike MDD, DT has a simple regular structure, which is easy to create. Therefore, the *static* approach starts with the creation of DT representing the desired function and then transforms it into MDD. To transform DT into MDD we apply the following steps on all levels of the DT in a bottom-up manner to eliminate redundant and duplicate nodes:

1. Remove all redundant nodes on the current level. Each edge incoming into a redundant node will now point to its single son.
2. Create a list of nodes for each group of duplicate nodes on the current level.
    a. Select and extract an arbitrary node from each list – these are the new unique nodes that will stay in the diagram.
    b. Each incoming edge into one of the nodes in any of the lists will now point to the node selected from the list.
3. If all levels have been processed, end, otherwise go to step 1.

The resulting MDD is ordered – it inherits the ordered property from the DT – and is also reduced, which is guaranteed by step 2. Fig. 3.11 shows an example of the transformation of a DT representing an integer function into MDD. Bold-outlined nodes mark a list of duplicate nodes and grayed nodes mark redundant nodes. Note that duplicate terminal nodes are kept for better readability.

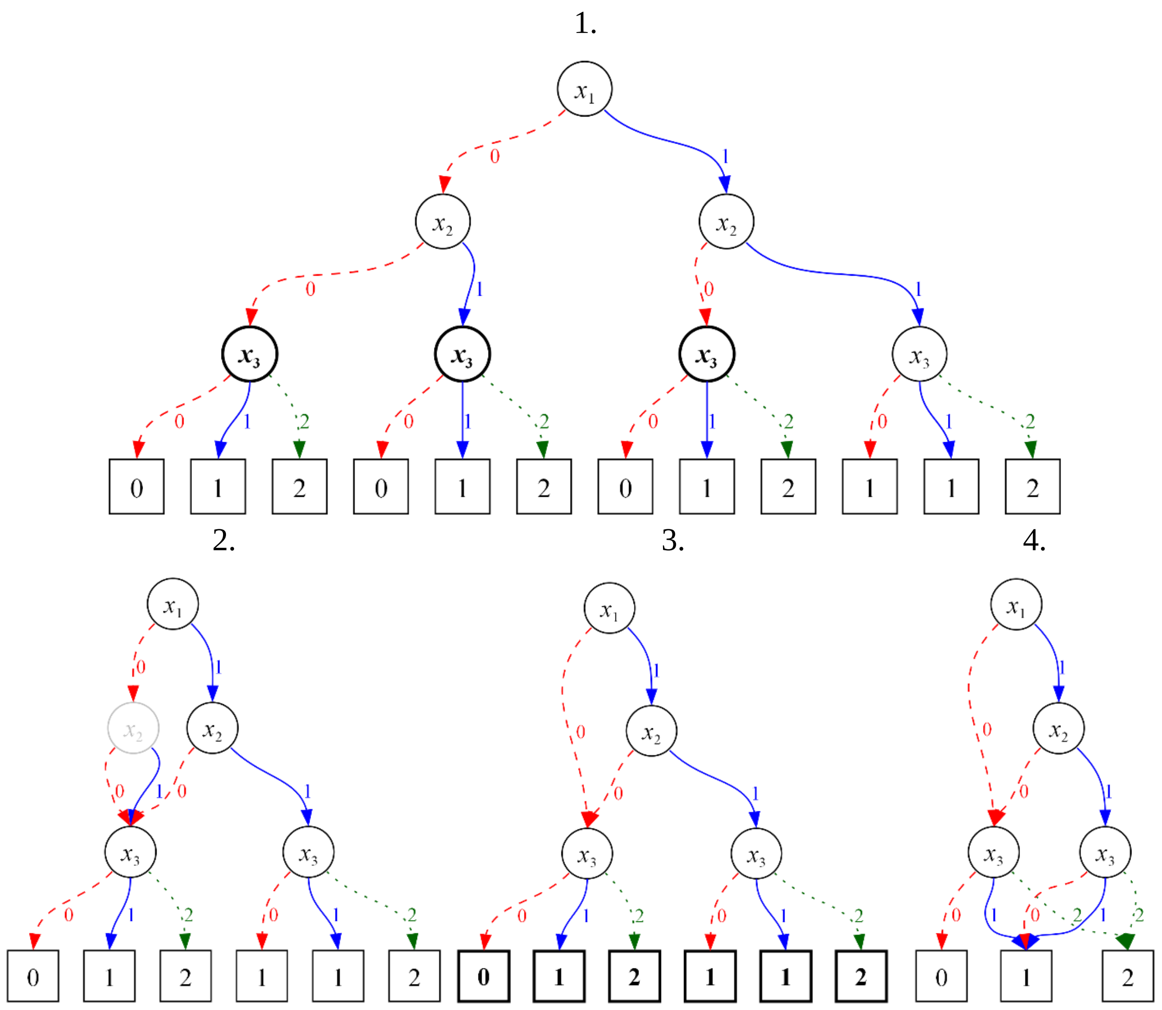


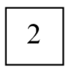
Fig. 3.11 Transformation of a DT into MDD by the gradual elimination of redundant and duplicate nodes (redundant terminal nodes are removed in the last step for better readability)

The above-described transformation follows Bryant's description of the *reduce* algorithm [11] for BDDs. Unfortunately, the *static* approach is inefficient because of the size of the initial DT – which is exponential in the number of variables. The process is also inefficient due to the considerable number of nodes it initially creates only to be subsequently removed. There exists a slightly better algorithm for static creation called *from-vector* [80]. The input of the algorithm is a truth vector (section 2.4.3) – practically the last level of the DT. The algorithm works in a similar bottom-up manner, but it avoids the creation of redundant and duplicate nodes. We provide a pseudocode of the *from-vector* algorithm in Alg. A.3.

The *static creation* is not practical for larger functions due to its exponential complexity. However, it may be useful for the creation of smaller functions that are easier to describe using a truth table (truth vector). Such a function can be further processed using the *dynamic* approach that we describe later in this section.

3.2.2.2 Direct Creation

Diagrams representing certain types of functions have a regular structure that we can utilize in the creation process. We call this function-specific approach the *direct* approach. The advantage of this approach is that it can create the diagram much faster than the general approaches. We have already encountered the direct approach in the creation of a terminal node (Alg. A.1) – representing a constant function – and the creation of an internal node (Alg. A.2). Fig. 3.12 shows an example of diagrams that can be easily created directly.



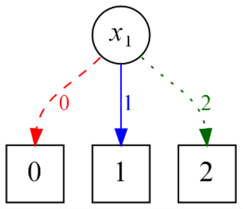


Fig. 3.12 Simple decision diagrams representing a constant function (left) and an integer function of a single variable (right)

3.2.2.3 Direct Creation of BDDs

3.2.2.3.1 Logical Conjunction and Disjunction

A very simple diagram type that we can create directly is BDD representing the function of logical conjunction (f) or logical disjunction (g) of n variables defined as:

$$f(\boldsymbol{x}) = x_1 \wedge x_2 \wedge \ldots \wedge x_n \tag{3.4}$$

$$g(\boldsymbol{x}) = x_1 \vee x_2 \vee \ldots \vee x_n. \tag{3.5}$$

Fig. 3.13 shows BDDs representing functions f and g respectively. Function (3.4) evaluates to 1 if and only if all variables have a value of 1 and to 0 otherwise. Similarly, function (3.5) evaluates to 1 if and only if at least one of the variables has a value of 1 and to 0 otherwise. This also follows from the fact that 0 is the absorbing element for the logical conjunction operation and 1 is the absorbing element for the logical disjunction operation. As the figure shows, BDDs elegantly capture this property. In each internal node, there is a possibility to go straight to a terminal node if the variable associated with a given node has a value equal to the absorbing element. We can also see that both BDDs have the same structure, the only difference is labels on the edges and values in the terminal nodes. Also, note, that variables in the conjunction and disjunction can be negated. In such a case the only modification is that we swap outgoing edges of the nodes.

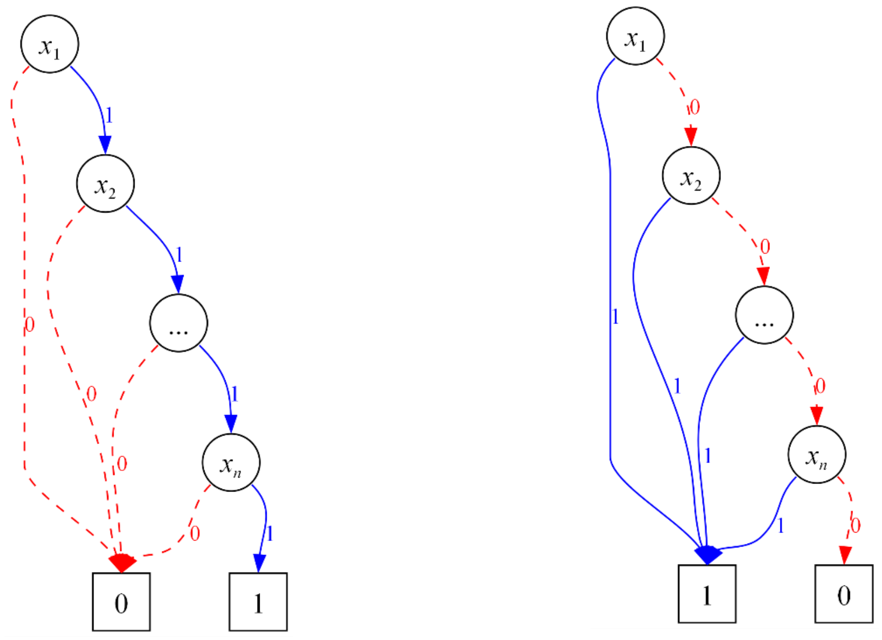


Fig. 3.13 BDDs representing logical conjunction (left) and logical disjunction (right) of n Boolean variables

Functions (3.4) and (3.5) agree with the definitions of series and parallel systems. Therefore, we can utilize the direct approach in the reliability analysis of such systems [81]. However, the more interesting use case is the representation of series-parallel systems. The creation of a decision diagram representing a series-parallel system is an exemplary case where we can utilize both the dynamic and the direct approach. During the creation, we dynamically merge directly created diagrams representing series and parallel parts of the system.

Disjunctive Normal Form (DNF) also known as Sum of Products (SoP) is a commonly used expression representation of the Boolean function [8]. In general, it consists of logical conjunctions joined by logical disjunctions. The following expression:

$$f(\boldsymbol{x}) = x_1\overline{x_2}x_3 \vee x_2\overline{x_3} \vee x_1x_2\overline{x_3}, \tag{3.6}$$

shows an example of DNF. To create a BDD for a function defined in the form of DNF we again can utilize both approaches by first directly creating a BDD for each product and subsequently dynamically merging them.

3.2.2.3.2 Parity Function

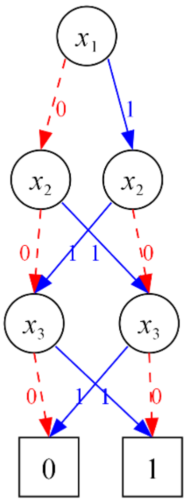
Another Boolean function that has regular representation in the form of BDD is a parity function of n variables [11]. The function is defined using the logical exclusive disjunction operation (XOR) in the following way:

$$f(\boldsymbol{x}) = x_1 \oplus x_2 \oplus \dots \oplus x_n. \tag{3.7}$$

It evaluates to 1 if and only if an odd number of variables has a value of 1 and to 0 otherwise. Fig. 3.14 shows BDD representing the odd parity function of 3 variables. Except for the variable at the root, each variable has exactly two nodes associated with it. As Bryant notes [11], the diagram has a ladder-like structure. In the figure, we can see that internal levels are identical for all variables (except the root one). Therefore, to create the diagram directly we add as many internal levels as necessary.

3.2.2.3.3 Structure k-out-of-n

A system with k-out-of-n structure is a specific system type that we have described in section 1.3.3. Structure function of such a system evaluates to 1 if and only if at least k variables have value of 1. BDD representing the structure function has a regular structure and, therefore, can be created directly, though, its structure is a bit more complicated. Parameters k and n influence the structure of the diagram. At the first $n-k+1$ levels of the diagram there exists a path starting at the first node on the left of the level that continues via 1-labeled edges of nodes in the path through the next $k-1$ levels ending in terminal node representing the value 1. Fig. 3.14 shows BDD representing 3-out-of-5 BSS (note that duplicate terminal nodes are kept for better readability). In the figure, we can see that on the left of the first 3 (in general $n-k+1$) levels a path starts that is terminated at the terminal node representing the value 1 containing 3 (k in general) 1-labeled edges.



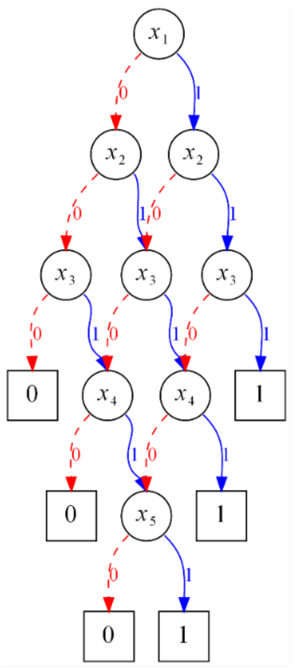


Fig. 3.14 BDD representing odd parity function of 3 variables (left) and BDD representing structure function of 3-out-of-5 BSS (right)

K-out-of-n system is a special case of a more general k-to-l-out-of-n system. Such a system is operation if at least k but no more than l components are operational. Let us not that such a system is an example of a noncoherent system (section 1.2.1). We can also directly create a BDD representing such a system [81], which has a similar just a bit more complicated structure.

3.2.2.3.4 Symmetric Functions

All the above-mentioned functions have a common property – they are symmetric functions. Therefore, it is no coincidence the BDDs representing the functions have a regular structure. In fact, it is a property of a reduced ordered diagram that when it represents a symmetric function of n variable it has some type of a regular structure that has at most $O(n^2)$ nodes [11].

3.2.2.4 Direct Creation of MDDs

3.2.2.4.1 Min and Max Functions

Just like with the functions of logical conjunction (3.4) and logical disjunction (3.5), we can also create MDDs representing their generalization – the min (f) and max (g) functions:

$$f(\boldsymbol{x}) = \min(x_1, x_2, \dots, x_n), \tag{3.8}$$

$$g(\boldsymbol{x}) = \max(x_1, x_2, \dots, x_n). \tag{3.9}$$

In Fig. 3.15, we can see MDDs representing integer functions (3.10) and (3.9), in the case when $m = 3$. Even though the structure looks a bit more complicated than the BDD counterparts, it is regular and can be created directly. Each internal level except the first level contains exactly $m - 1$ internal nodes. At each level the edges representing the absorbing value of the operation lead directly to the terminal node and the rest of the edges lead to the nodes on the next level. In general, there are $n * m - 1$ internal nodes in the diagram.

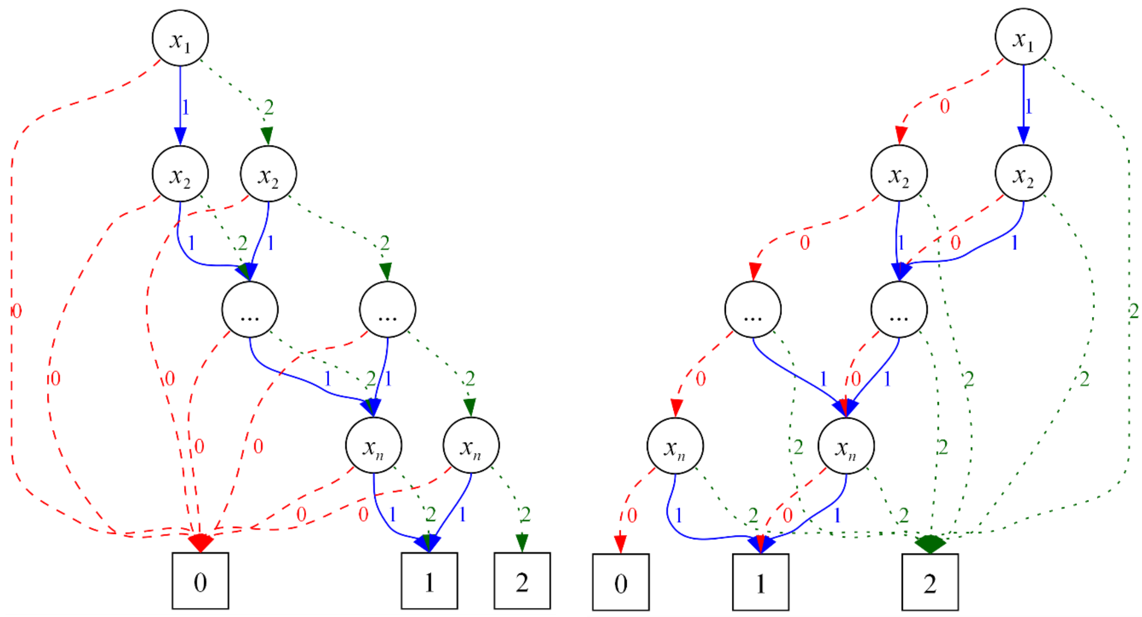


Fig. 3.15 BDDs representing the min function (left) and the max function (right) of n integer variables

3.2.2.4.2 Structure k-out-of-n

Construction of MDDs representing various types of multi-state k-out-of-n systems follows the same ideas as the simpler case of the BSS (section 3.2.2.3.3). However, their structure is more complicated – it consists of layers of structures like the structure depicted in Fig. 3.14 [82] i.e., the regular structure of the diagram can be seen in a three-dimensional layout. Furthermore, there also exist a few special cases such as k-out-of-$(2k - 1)$ systems [83], which can be represented by an MDD with a regular structure.

3.2.2.5 Dynamic Creation

The static and direct approaches are useful in the creation of specific functions. However, they are not suitable for general diagram creation. For that, we use the so-called *dynamic* approach, which utilizes Shannon's expansion described in section 3.1.2. The principal idea is to first split the function into multiple simpler functions joined with binary operations. Then we start by directly or statically creating diagrams for the simpler functions. After that, we continue by gradually merging those diagrams into more complicated ones. In the end, we are left with a single diagram representing the desired function.

3.2.2.5.1 Apply

The merger of two diagrams can be realized using a recursive algorithm called *apply* introduced for BDDs by Bryant [11][84] and later generalized for MDDs [12]. The algorithm uses the following two recursive relations derived from Shannon's expansions. For two Boolean functions f and g and Boolean binary operation $\odot$ it holds that [11]:

$$(f \odot g)(\boldsymbol{x}) = x_i\big(f(1_i,\boldsymbol{x}) \odot g(1_i,\boldsymbol{x})\big) \vee \overline{x_i}\big(f(0_i,\boldsymbol{x}) \odot g(0_i,\boldsymbol{x})\big), \qquad (3.10)$$

and generalized for two integer functions f and g and binary operation $\odot$ it holds that [12]:

$$(f \odot g)(\boldsymbol{x}) = \sum_{k=0}^{m_i-1} \Big(\{x_i \leftrightarrow k\} * \big(f(k_i,\boldsymbol{x}) \odot g(k_i,\boldsymbol{x})\big)\Big). \qquad (3.11)$$

The input of the *apply* algorithm is two diagrams and a binary operation $\odot$ closed on the codomain of the function i.e., of the following form:

$$\{0,1,\dots,m-1\}^2 \to \{0,1,\dots,m-1\}, \qquad (3.12)$$

The algorithm is invoked on the roots of the diagrams. In each step, it visits a pair of nodes – one from each diagram – and either creates an internal or terminal node. The terminal node can be created in two situations:

1. When both nodes are terminal in which case the value in the new node is determined by *applying* the binary operation $\odot$ on the values represented by the nodes.
2. When one of the nodes is a terminal node representing an absorbing element of the binary operation $\odot$. Then, the value in the new node is the absorbing element.

The creation of the terminal node is the terminating case for the recursion.

Most of the time, the nodes that enter the step are both internal nodes (or one of them is a terminal representing a non-absorbing element). Let A and B be nodes that entered the step. Let us assume without loss of generality that $i_A < i_B$. If one of the nodes is terminal, we proceed as if it had the index equal to its level in which case it will certainly be greater than the other index. The result of the step is a new internal node associated with a variable x_{i_A}. The m_{i_A} sons of the new node are obtained with a recursive call of the step with a pair (A_k, B) for $k = 0,1,\dots,m_{i_A}-1$. Alg. A.4 presents complete pseudocode.

An essential part of the *apply* algorithm is to avoid processing the same pair of nodes multiple times. If this is satisfied the complexity of the algorithm is $O(\mathscr{s}_1 * \mathscr{s}_2)$ where $\mathscr{s}_1, \mathscr{s}_2$ are the sizes of the input diagrams. This can be achieved by maintaining a cache table with a pair of pointers to nodes that entered the step of the algorithm as key and a pointer to the node created in the step as value. In each step the cache table is first queried, and if there already is an entry for the current pair of nodes that node is returned as the result of the step.

The algorithm can be made more efficient thanks to the node sharing and immutability of nodes (section 3.2.1). The improvement can be made by following the observation that if node C is the result of step of the apply call with nodes A, B and operation $\odot$ it will also always be the result in the future if the step is called with the same arguments. Thus, it is beneficial to maintain the cache table globally – shared for all *apply* calls. The cache requires a small adjustment in this case. Since *apply* works for any binary operation, the operation (unique integer ID) must be part of the key. Therefore, a cache query could have the following form $\text{CONTAINS}\left(applyCache, \left(left, right, \text{GETID}(\odot)\right)\right)$. Since many binary operations are *commutative*, further improvement can be made by adjusting the cache table in such a way that keys $\left(left, right, \text{GETID}(\odot)\right)$ and $\left(right, left, \text{GETID}(\odot)\right)$ map to the same value. For instance, if we use a hash table to implement the cache, we can utilize some symmetric function (e.g. bitwise exclusive OR) to combine hashes $\text{HASH}(left)$ and $\text{HASH}(right)$ and adjust the equality comparison so that it compares the key triplets as sets (i.e. not considering the order of elements). Finally, the last aspect of caching to consider is the size of the cache table, which could grow significantly for larger diagrams. Because of this, many implementations use a fixed size for the table. For example, a hash table may resolve collisions by overwriting existing entries. This can result in re-computation of some results but provides a reasonable tradeoff with the memory complexity of the algorithm.

Fig. 3.16 shows an example of the *apply* algorithm in the merger of two BDDs using the logical conjunction. Nodes of the input diagrams are marked using upper-case letters, and the nodes of the resulting diagram are marked using a pair of letters, which signifies nodes from the input diagrams that were processed in the step that created the resulting node.

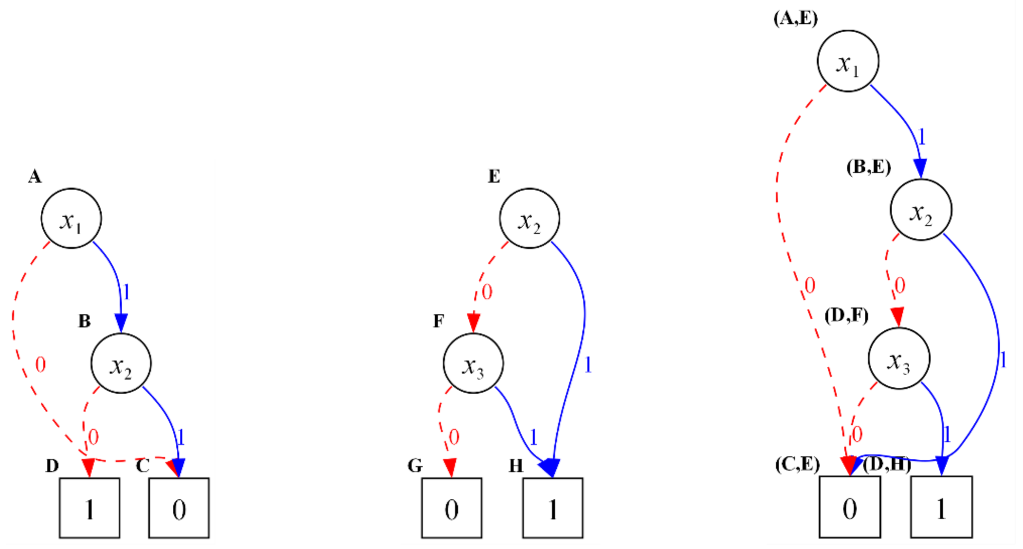


Fig. 3.16 Merger of two BDDs representing Boolean functions $f(\boldsymbol{x}) = x_1\overline{x_2}$ (left) and $g(\boldsymbol{x}) = x_2 \vee x_3$ (middle) using the *apply* algorithm with logical conjunction

3.2.2.5.2 ITE and CASE

There exists an alternative to the *apply* algorithm for Boolean functions – the *If-Then-Else* (*ITE*) operator [61]. *ITE* is a ternary Boolean operator defined as:

$$\textsc{Ite}(A, B, C) = \textbf{if } A \textbf{ then } B \textbf{ else } C, \tag{3.13}$$

where A, B, and C are Boolean functions (nodes of a diagram). Like *apply*, using the *ITE* operator we can create BDD for any function by merging BDDs of simpler functions. Also, like *apply*, the merger of diagrams using *ITE* is a recursive procedure. However, unlike *apply*, the *ITE* operator does not take a Boolean binary operation as its input. Instead, all Boolean operations can be defined in terms of the *ITE* operator [60]. Tab. 3.3 contains the definitions for common Boolean operations.

The *ITE* operator is limited to BDDs and Boolean functions. A generalization called *CASE* exists that can manipulate MDDs and integer functions. CASE is a $(m+1)$-ary operation defined in a way similar to the generalized Shannon's expansion (3.2) as [12]:

$$\mathrm{CASE}(A, B_1, B_2, \dots, B_{m-1}) = \sum_{k=0}^{m-1} (\{A \leftrightarrow k\} * B_k). \tag{3.14}$$

Just like with the *ITE* operator, we can use the *CASE* operator to define common operations such as min and max [85] for an m-valued logic function. In Tab. 3.4 we show the definitions for the 4-valued logic function presented in [85]. Generalization to m-valued logic is relatively straightforward, thou not as simple as using the *apply* operation.

Tab. 3.3 Definitions of common Boolean operations in terms of the *ITE* operator

Name	**Expression**	**ITE form**
Logical negation	$\bar{a}$	$\mathrm{ITE}(a, 0,1)$
Logical conjunction	ab	$\mathrm{ITE}(a, b, 0)$
Negated logical conjunction (NAND)	$\overline{ab}$	$\mathrm{ITE}(a, \bar{b}, 1)$
Logical disjunction	$a \vee b$	$\mathrm{ITE}(a, 1, b)$
Negated logical disjunction (NOR)	$\overline{a \vee b}$	$\mathrm{ITE}(a, 0, \bar{b})$
Exclusive logical disjunction (XOR)	$a \oplus b$	$\mathrm{ITE}(a, \bar{b}, b)$

Tab. 3.4 Definitions of min and max operations in 4-valued MLV using the *CASE* operator

Name	**Expression**	**CASE form**
Minimum	$\min(A, B)$	$\mathrm{CASE}(A, 0, \mathrm{CASE}(B, 0,1,1,1), \mathrm{CASE}(B, 0,1,2,2), B)$
Maximum	$\max(A, B)$	$\mathrm{CASE}(A, B, \mathrm{CASE}(B, 1,1,2,3), \mathrm{CASE}(B, 2,2,2,3), 3)$

3.2.3 Extended Apply

The basic apply algorithm described in section 3.2.2 accepts any *binary* operation of the form (3.12) as its input. However, in some situations, the functions that we need to represent contain d-ary operators. As an example, let us consider a simple logic circuit depicted in Fig. 3.17, which implements the following Boolean function:

$$f(\boldsymbol{x}) = x_1 x_2 x_3 \vee x_4. \tag{3.15}$$

If we wanted to create BDD representing function (3.15), using the *apply* algorithm, it would be called in the following way:

$$\textsc{Apply}(\textsc{Apply}(\textsc{Apply}(X_1, X_2, \wedge), X_3, \wedge), X_4, \vee),$$

where symbol X_i represents BDD representing variable x_i. However, since the AND gate that realizes the logical conjunction is a three-input gate, it would be more convenient and descriptive to use the apply algorithm in the following way:

$$\textsc{Apply}(\textsc{Apply}(X_1, X_2, X_3, \wedge), X_4, \vee)$$

i.e., to be able to call apply with three input diagrams instead of two.

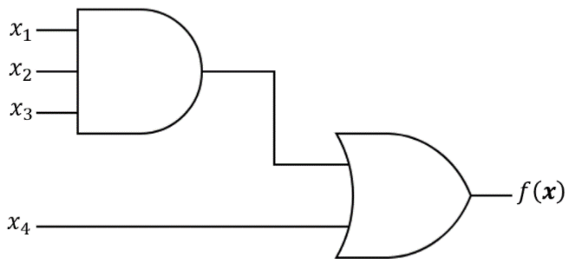


Fig. 3.17 Simple combinatorial circuit with four inputs and one output

Any series of applications of binary *associative* operation $\odot$ can be easily extended to a single d-ary operation denoted as $\odot_d$ in the following way:

$$\begin{aligned} & f_1 \odot \big(f_2 \odot (\dots (f_{d-1} \odot f_d) \dots)\big) \\ & = f_1 \odot f_2 \odot \dots \odot f_{d-1} \odot f_d \\ & = \odot_d\, (f_1, f_2, \dots, f_{d-1}, f_d). \end{aligned} \tag{3.16}$$

The d-ary version $\odot_d$ of the operation $\odot$ is simply defined in terms of multiple applications of the binary version of the operation. Using the notation $\odot_d$, we can also express the relation (3.16) as the following recurrent relation:

$$\begin{aligned}
&\odot_d\,(f_1, f_2, \dots, f_{k-1}, f_d)\\
&=\odot_{d-1}\left(f_1, f_2, \dots, \odot_2\,(f_{d-1}, f_d)\right)\\
&=\odot_{d-2}\left(f_1, f_2, \dots, \odot_2\left(f_{d-2}, \odot_2\,(f_{d-1}, f_d)\right)\right)\\
&= \cdots\\
&=\odot_2\left(f_1, \odot_2\left(f_2, \odot_2\left(\dots, \odot_2\left(f_{d-2}, \odot_2\,(f_{d-1}, f_d)\right) \dots\right)\right)\right).
\end{aligned} \tag{3.17}$$

Furthermore, we can express the relation (3.11) that is a key part of the *apply* operation as follows:

$$\begin{aligned}
&\odot_2\,(f_1, f_2)(\boldsymbol{x})\\
&=\odot_2\left(f_1(\boldsymbol{x}), f_2(\boldsymbol{x})\right)\\
&= \sum_{k=0}^{m_i-1} \{x_i \leftrightarrow k\} * \left(\odot_2\left(f_1(k_i, \boldsymbol{x}), f_2(k_i, \boldsymbol{x})\right)\right).
\end{aligned} \tag{3.18}$$

If expressions $f_1, f_2, \dots, f_d$ in formula (3.17) are functions of the same variables defined by Boolean vector $\boldsymbol{x}$, the recurrent relationship defined by formula (3.17) can be transformed into the following relationship:

$$\begin{aligned}
&\odot_d\,(f_1, f_2, \dots, f_{d-1}, f_d)(\boldsymbol{x})\\
&=\odot_d\left(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \dots, f_{d-1}(\boldsymbol{x}), f_d(\boldsymbol{x})\right)\\
&=\odot_{d-1}\left(f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \dots, \odot_2\left(f_{d-1}(\boldsymbol{x}), f_d(\boldsymbol{x})\right)\right)\\
&= \cdots\\
&=\odot_2\left(f_1(\boldsymbol{x}), \odot_2\left(\dots, \odot_2\left(f_{d-1}(\boldsymbol{x}), f_d(\boldsymbol{x})\right) \dots\right)\right)\\
&=\odot_2\left(f_1(\boldsymbol{x}), \odot_2\,(\dots, \odot_2\,(f_{d-1}, f_d) \dots)(\boldsymbol{x})\right).
\end{aligned} \tag{3.19}$$

By combining this formula with formula (3.18), we obtain:

$$\begin{aligned}
&\odot_d\,(f_1, f_2, \dots, f_{d-1}, f_d)(\boldsymbol{x})\\
&= \cdots\\
&=\odot_2\left(f_1(\boldsymbol{x}), \odot_2\,(\dots, \odot_2\,(f_{d-1}, f_d) \dots)(\boldsymbol{x})\right)\\
&= \sum_{k=0}^{m_i-1} \{x_i \leftrightarrow k\} * \left(\odot_2\left(f_1(k_i, \boldsymbol{x}), \odot_2\,(\dots, \odot_2\,(f_{d-1}, f_d) \dots)\right)(k_i, \boldsymbol{x})\right)\\
&= \sum_{k=0}^{m_i-1} \{x_i \leftrightarrow k\} * \left(\odot_2\left(f_1(k_i, \boldsymbol{x}), \odot_2\left(\dots, \odot_2\left(f_{d-1}(k_i, \boldsymbol{x}), f_d(k_i, \boldsymbol{x})\right) \dots\right)\right)\right).
\end{aligned} \tag{3.20}$$

Finally, according to (3.16), nested parentheses in (3.20) can be removed by replacing the prefix operator $\odot_2$ by the infix version $\odot$. This results in an extended version of the relation (3.11):

$$\odot_d\,(f_1, f_2, \dots, f_d)(\boldsymbol{x}) = \sum_{k=0}^{m_i-1} \Big(\{x_i \leftrightarrow k\} * \big(\odot_d\,(f_1, f_2, \dots, f_d)(k_i, \boldsymbol{x})\big)\Big). \tag{3.21}$$

Using this relation, we present a novel *extended* version of the *apply* algorithm, which accepts d-tuple of diagrams and d-ary operation. The algorithm is one of the contributions of this thesis. The new version is similar to the *apply* algorithm described in section 3.2.2. The key difference is that in each step it visits d-tuple of nodes, one from each diagram. Alg. 3.1, Alg. 3.2, and Alg. 3.3 contains the complete pseudocode of the *extended apply*. In the pseudocode, functions like ROOT, VALUE, INDEX, or the auxiliary GETSON applied on the tuple operate on each element of the tuple separately returning a tuple of individual results.

```
procedure EXTENDEDAPPLY(($D_1$, $D_2$, ..., $D_d$), $\odot_d$)
    root ← EXTENDEDAPPLYSTEP(ROOT(($D_1$, $D_2$, ..., $D_d$)), $\odot_d$)
    return MDD(root)
end procedure
```

Alg. 3.1 Entry point of the extended apply algorithm

```
procedure EXTENDEDAPPLYSTEP(($N_1$, $N_2$, ..., $N_d$), $\odot_d$)
    if CONTAINS(applyCache, ($N_1$, $N_2$, ..., $N_d$)) then
        return LOOKUP(applyCache, ($N_1$, $N_2$, ..., $N_d$))
    end if
    node ← NULL
    if ALLTERMINAL(($N_1$, $N_2$, ..., $N_d$)) then
        node ← CREATETERMINALNODE($\odot_d$(VALUE(($N_1$, $N_2$, ..., $N_d$))))
    else if ANYABSORBING($\odot$, ($N_1$, $N_2$, ..., $N_d$)) then
        node ← CREATETERMINALNODE(ABSORBINGELEMENT($\odot$))
    else
        i ← min(LEVEL(($N_1$, $N_2$, ..., $N_d$)))
        sons ← MAKETUPLE($m_i$ )
        for k = 0 to $m_i$ do
            sons[k] ← EXTENDEDAPPLYSTEP(GETSON(i, ($N_1$, $N_2$, ..., $N_d$), k), $\odot_d$)
        end for
        node ← CREATEINTERNALNODE(i, sons)
    end if
    PUT(applyCache, ($N_1$, $N_2$, ..., $N_d$), node)
    return node
end procedure
```

Alg. 3.2 Recursive step of the extended apply algorithm

```
procedure GETSON($i$, $node$, $k$)
    if INDEX($node$) = $i$ then
        return SON($node$ , $k$)
    else
        return $node$
    end if
end procedure
```

Alg. 3.3 Helper function used in the step of the extended apply algorithm

The *extended apply* algorithm gives the same results as multiple calls of the basic *apply* algorithm. Its main advantage is the convenience – a single call to the *extended apply* can replace multiple nested calls to the basic *apply*. On the other hand, its disadvantage may lie in its complexity, which is $\mathrm{O}(\mathscr{s}_1 * \mathscr{s}_2 * \ldots * \mathscr{s}_d)$, since, in the worst case, we process each d-tuple of nodes.

The big-O notation sets the upper bound on the number of steps of the algorithm, which may not necessarily reflect the real performance of the algorithm. Therefore, an experimental comparison of the *apply* and *extended apply* is an interesting task. Furthermore, the comparison may also provide insight into the feasibility of the *extended apply* since its heavy utilization of the recursion may be limited by the default size of the call stack.

3.2.4 Diagram Manipulation

Diagrams can be queried, evaluated, and manipulated in various ways. In this section, we describe selected diagram algorithms that we use later in the description of algorithms for reliability analysis.

3.2.4.1 Satisfy-count

The first algorithm that we describe is called *satisfy-count* introduced by Bryant [11] for BDD. We describe a version generalized to MDD. *Satisfy-count* is a simple query on the diagram that returns the number of satisfying variable assignments i.e., if the diagram represents a function f the algorithm returns the number of input vectors $\boldsymbol{x}$ such that $f(\boldsymbol{x}) = j$ where the diagram and the value j are parameters of the algorithm.

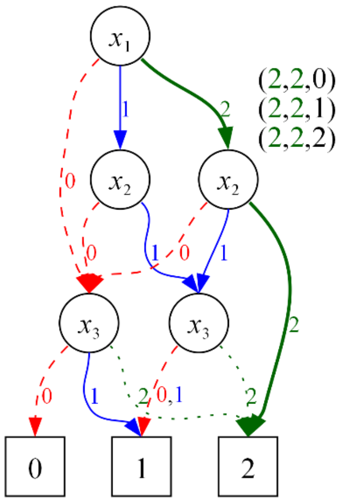


Fig. 3.18 Evaluation of a function represented by a decision diagram

The pseudocode of the algorithm is presented in Alg. A.5. To understand the rationale of the algorithms first let us consider a simpler algorithm that only counts the number of *paths* from the root node to the terminal node containing the value j. Each internal node on a path from the root node to a terminal node represents all possible values of the variable x_i. Therefore, the number of paths starting at a given internal node can be calculated as the sum of the number of paths starting at each son of the node. However, an edge can skip over some levels and therefore a single path can correspond to multiple input vectors, as we can see in Fig. 3.18. In the figure, the highlighted path corresponds to three state vectors. Thus, to account for the skipped levels, we need to multiply the number obtained from the son by the number of vectors corresponding to skipped levels. In the pseudocode, this number is calculated by the DOMAINPRODUCT function.

3.2.4.2 Cofactor

We have encountered cofactor (3.2) in the definition of the decision diagram and the *apply* algorithm for its dynamic creation. However, in neither of these situations, we were required to calculate the cofactor of a function directly. Nevertheless, the calculation of the cofactor of a function is an essential step in various algorithms – for example, in the calculation of logic derivatives described further in section 3.3.4.

The *cofactor* algorithm for MDD is a generalization of the *restrict*[2] algorithm proposed by Bryant [11] for BDD. Computation of cofactor $f(a_i, \boldsymbol{x})$ is a simple transformation of the diagram. The core idea is to remove all internal nodes associated with variable x_i and redirect all edges ending in such nodes into its a^{th} son (where $a \in \{0,1,\dots,m_i-1\}$). However, the operation must maintain the invariants of node uniqueness and node immutability. Therefore, the algorithm produces a new diagram representing the cofactor without altering the original diagram. The new diagram, however, may share many nodes with the original. The pseudocode of the algorithm is presented in Alg. A.6.

In Fig. 3.19, we can see an example of MDD representing function $f(\boldsymbol{x})$ and MDD representing the cofactor $f(2_1, \boldsymbol{x})$. The grey node in the right part of the image highlights the difference between the original MDD and its cofactor.

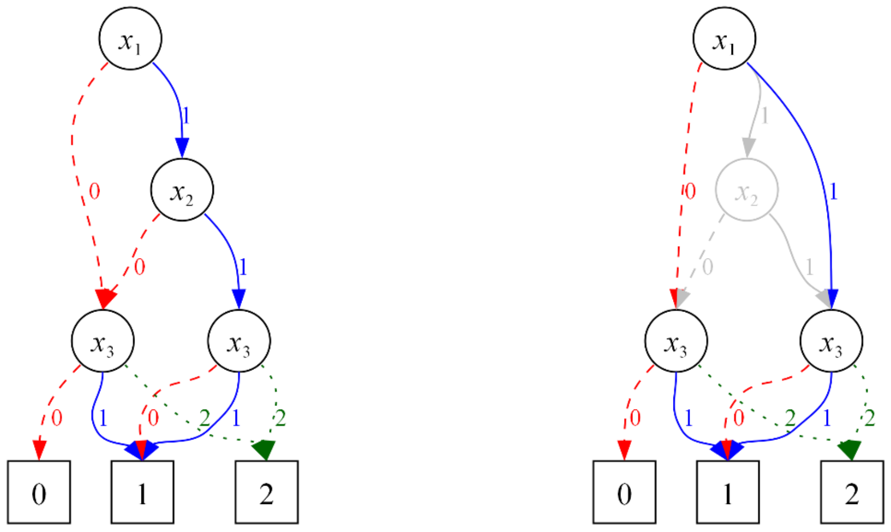


Fig. 3.19 MDD representing function $f(\boldsymbol{x})$ (left) and MDD representing cofactor $f(2_1, \boldsymbol{x})$

The basic version of the *cofactor* algorithm presented in the pseudocode fixes the value of only one variable at a time. However, in some situations, we need to fix the value of multiple variables. This can be achieved by simply using the *cofactor* algorithm multiple times. Unfortunately, such an approach would result in repetitive re-computations. A much better approach is to generalize the *cofactor* algorithm so that it accepts a list of pairs (i, a). The generalized version works in a very similar way with the difference that it "skips" nodes on multiple levels – specified in the list of pairs.

[2] Cofactor of a function is also known as restriction of a function in some literature – hence the name of Bryan's algorithm

3.2.4.3 Transform

Another algorithm that we utilize in the calculation of logic derivatives is the *transform* algorithm. This algorithm operates on the values of the function stored in the terminal nodes, transforming them using function γ of the following form:

$$\gamma(a): \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}, \tag{3.22}$$

where $a \in \{0,1,\dots,m-1\}$. Just like the *cofactor* algorithm, the *transform* algorithm's core part is a recursive step. The pseudocode of the algorithm is presented in Alg. 3.4 and Alg. 3.5. Implementation of the step is rather simple. When it visits an internal node it simply recurses deeper into the diagram and afterward creates a new node. The transformation happens when the step visits a terminal node A in which case its returns new terminal node representing value $\gamma(A)$.

The *transform* algorithm can be used to implement various unary operations [9] such as complement, successor, or predecessor. For example, to define the complement of an integer function we would use the function γ_1 defined as:

$$\gamma_1(a) = m - 1 - a. \tag{3.23}$$

Fig. 3.20 depicts an example of an MDD representing an integer function and an MDD representing the complement of the function obtained using the *transform* algorithm. Another frequent use case is when we need to narrow the codomain of a function – for instance, to transform it into a pseudo-logic function (2.5). In such a case, we may use the following γ_2 function (possibly with any other relational operator):

$$\gamma_2(a) = \begin{cases} 1, & a \geq j \\ 0, & \text{otherwise,} \end{cases} \tag{3.24}$$

where $j \in \{0,1,\dots,m-1\}$.

Finally, the last, less obvious, use case that we present is in the implementation of the *reduce* algorithm [11]. The reduce algorithm transforms any MDD that contains duplicate or redundant nodes into canonical form by "removing" all such nodes. The algorithm can be implemented using the transformation with the *identity* function defined as:

$$\gamma_3(a) = a. \tag{3.25}$$

The resulting diagram contains no duplicate and no redundant nodes thanks to the factory functions (Alg. A.1, Alg. A.2), since each node is re-created by the functions – which only produce unique nodes.

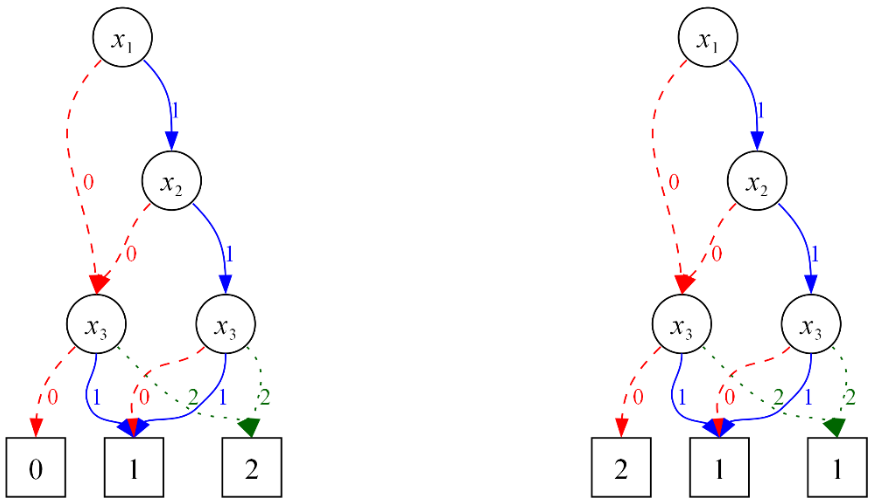


Fig. 3.20 MDD representing an integer function (left) and MDD representing complement of the function (right)

Let us note that the same transformation of an MDD can be achieved by using the *apply* algorithm with a binary operation and an MDD representing a constant function with a value equal to the neutral element of the binary operation. However, we consider such a solution less efficient and less readable. Thus, we present the pseudocode directly in this section although the algorithm itself and ideas it utilizes are not exactly novel.

```
procedure TRANSFORM(diagram, γ)
   root ← ROOT(diagram)
   newRoot ← TRANSFORMSTEP(root, γ)
   return MDD(newRoot)
end procedure
```

Alg. 3.4 Entry point of the transform algorithm

```
procedure TRANSFORMSTEP(node, γ)
   if ISTERMINAL(node) then
      return CREATETERMINALNODE(γ(VALUE(node)))
   end if
   if CONTAINS(memo, node) then
      return LOOKUP(memo, node)
   end if
   i ← INDEX(node)
   sons ← MAKETUPLE($m_i$)
   for k = 0 to $m_i$ do
      oldSon ← SON(node, k)
      sons[k] ← TRANSFORMSTEP(oldSon, γ)
   end for
   newNode ← CREATEINTERNALNODE(j, sons)
   PUT(memo, node, newNode)
   return newNode
end procedure
```

Alg. 3.5 Recursive step of the transform algorithm

3.2.4.4 General Diagram Manipulation

When we analyze the pseudocodes of the *cofactor* and *transform* algorithms, we may notice that their step has a similar structure – which is no coincidence. As a matter of fact, we would see a similar structure in the implementation of other algorithms as well. In Alg. 3.6 we present a pseudocode that tries to capture a general structure of a recursive diagram transforming the algorithm.

In the pseudocode, there are a few conditions that terminate the recursion. The first one (denoted as (a)) deals with memoization, which we address below. The second one (denoted as (b)) identifies a situation that does not require further evaluation – the identification of the situation is checked by the function NoNeedToContinue. An example of such a situation can be found in the *cofactor* algorithm (Alg. A.6). The third point (denoted as (c)) is checked by the NeedsProcessing function, which identifies nodes that require some transformation, which is expressed by the Process function. At this point, the processing function can terminate the recursion (as we do in the pseudocode) or the processing of the diagram may continue – depending on the specifics of the algorithm.

Finally, let us note the presented general algorithm does a transformation of the node that entered the step. However, the result of the step does not necessarily need to modify the node (more precisely, to return a new node). It can also just compute and return some value like the *satisfy-count* algorithm (Alg. A.5) does.

```
procedure GeneralStep(node, ...)
    if Contains(memo, node) then                    ▷ (a)
        return Lookup(memo, node)
    end if
    if NoNeedToContinue(node) then                  ▷ (b)
        return node
    end if
    if NeedsProcessing(node) then                   ▷ (c)
        return Process(node)
    end if
    i ← Index(node)
    sons ← MakeTuple($m_i$)
    for $k = 0$ to $m_i$ do
        oldSon ← Son(node, k)
        sons[k] ← GeneralStep(oldSon, ...)
    end for
    newNode ← CreateInternalNode(j, sons)
    Put(memo, node, newNode)
    return newNode
end procedure
```

Alg. 3.6 General structure of a step of a recursive diagram manipulation algorithm

3.2.4.4.1 Result Memoization

Considering the efficiency of the algorithm utilizing the step, the essential part of the step is the *memoization* of the results. Since the nodes are immutable, the step must always return the same result for the same input node[3]. The step of the algorithm may try to visit some nodes more than once because of the node sharing. Consequently, to avoid expensive recomputations that would necessarily lead to the same result we maintain a lookup table of computed results – a *memo*. A key to the table is a pointer to the input node and the value is either a pointer to a new node or some numeric value – depending on the specifics of the actual algorithm. Our library TeDDy also implements an alternative to the lookup table – it uses the nodes themselves to store the results of the computation at the given node.

The memoization ensures that each node is processed at most once. Therefore, the computation complexity of the algorithm is $O(\mathscr{s} * \mathfrak{o} * \mathscr{t})$, where $\mathscr{s}$ is the number of nodes in the diagram, $O(\mathfrak{o})$ is the complexity of the node processing, and $O(\mathscr{t})$ is the complexity of querying the table. Typically, the complexity of the node processing operation is $O(1)$. The complexity of the table query depends on the implementation of the table. For example, when a hash table is used, the complexity is also $O(1)$. Our approach of storing the memoized result in the nodes also has complexity $O(1)$. Consequently, the complexity of the algorithm is practically $O(\mathscr{s})$.

3.2.4.4.2 Order of Variables

In all pseudocodes, we assumed – for simplicity – that the diagram uses the default order of variables. However, when we consider other orders of variables, we need to differentiate $\textsc{Index}(A)$ and $\textsc{Level}(A)$. A possible approach is to also store the level of a node as its property. However, such an approach would use more memory than necessary. Since there is a close relation between the level and index of nodes on that level, it is sufficient to maintain two mappings – level-to-index and index-to-level, which can be simply implemented as arrays where the index corresponds to level and index respectively.

[3] With the assumption that the step is not part some randomized algorithm – neither of the algorithms that we consider in the thesis involve randomness.

3.3 Decision Diagrams in Reliability Analysis

The structure function is an integral part of the reliability analysis. Definitions (1.1), (1.2), and (1.3) of the structure function agree with definitions (2.2), (2.3), and (2.4) of different discrete functions. Thus, an obvious application of decision diagrams is the representation of the structure function. In this section, we describe the evaluation of selected reliability characteristics of a system using the structure function represented by a decision diagram.

3.3.1 Structure Function Representation

The typical approach to the representation of the structure function is to represent the function with a single MDD. In section 3.2.2 and section 3.2.3, we have presented various approaches to the construction of the MDD. The diagram construction is a crucial aspect of the reliability analysis of complex systems, because of the considerable size of the diagram.

Besides the straightforward approach to the structure function representation, there exists an alternative option for the description of MSS. The idea is to describe each system state individually using a pseudo-logic function [86]. For example, let us consider a structure function $\phi(\boldsymbol{x})$ describing a 3-state MSS. Then we can use two functions $\phi(\boldsymbol{x}) \geq 1$ and $\phi(\boldsymbol{x}) \geq 2$ to fully describe the system. In general, for an m-state MSS, we need to use $m-1$ functions describing states $1, 2, \dots, m-1$. In Fig. 3.21 we can see an example of a structure function describing 3-state MSS represented using both approaches.

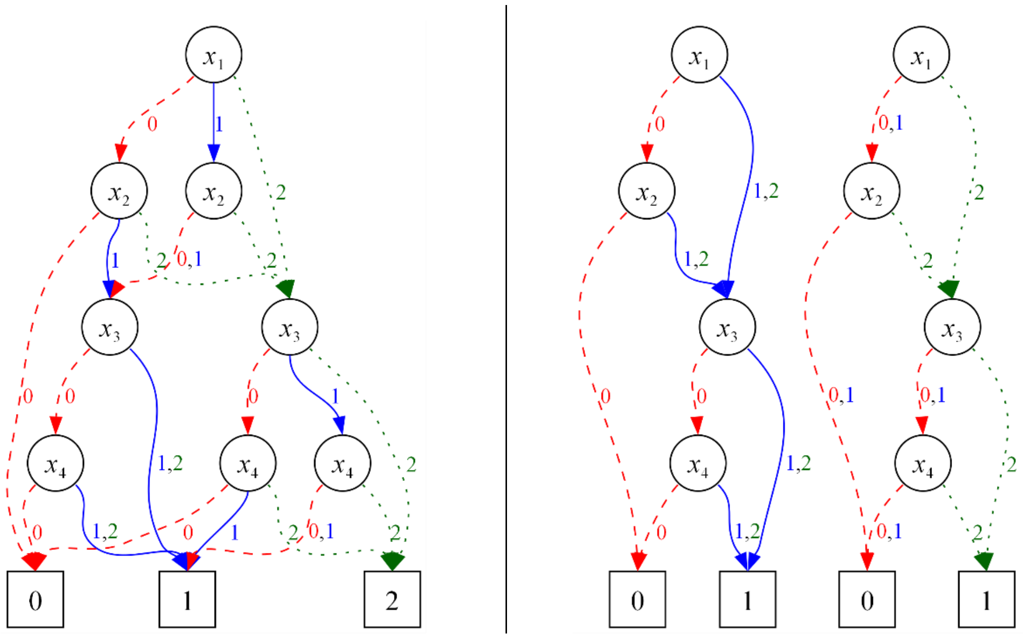


Fig. 3.21 Structure function $\phi(\boldsymbol{x})$ represented using a single diagram (left) and a series of diagrams (right) representing functions $\phi(\boldsymbol{x}) \geq 1$ and $\phi(\boldsymbol{x}) \geq 2$ respectively (right)

An interesting question is whether one of the approaches is more efficient with regard to the number of unique nodes in the diagrams or the speed of algorithms operating on the diagrams.

3.3.2 Topological Analysis

System state frequency $Fr^{=j}$ (1.9) is a simple topological characteristic of a system. It is defined as the relative number of state vectors for which the system described by structure function ϕ is in state j:

$$Fr^{=j} = \frac{\alpha_{\phi,j}}{\alpha_{\phi}}, \tag{3.26}$$

where α_{ϕ} denotes the total number of state vectors i.e., the size of the domain of the function (Tab. 2.1), and $\alpha_{\phi,j}$ denotes the number of state vectors $\boldsymbol{x}$ such that $\phi(\boldsymbol{x}) = j$. Therefore, to calculate the state frequency we need to calculate the numbers $\alpha_{\phi,j}$ and α_{ϕ}. α_{ϕ} can be calculated by simply multiplying the domains of all variables:

$$\alpha_{\phi} = \sum_{i=1}^{n} m_i, \tag{3.27}$$

which simplifies to:

$$\alpha_{\phi} = m^n, \tag{3.28}$$

for the m-valued logic function.

A straightforward approach to the calculation of $\alpha_{\phi,j}$ would be to evaluate the system for each possible state vector and count the number of *satisfying* input vectors. However, such a naive approach would be computationally infeasible even for tens of variables.

Evaluation of the diagram is a simple traversal from the root to a terminal node as shown in Fig. 3.18. In general, the number of state vectors corresponding to a path $\mathscr{p}$ can be calculated using the following formula:

$$\alpha_{\mathscr{p}} = \prod_{i \in \mathcal{I}'_{\mathscr{p}}} m_i, \tag{3.29}$$

where $\mathcal{I}'_{\mathscr{p}}$ is the set of indices of variables that are not present in path $\mathscr{p}$. Therefore, an improvement to the calculation can be made by using the following formula:

$$\alpha_{\phi,j} = \sum_{\mathscr{p}}^{\mathcal{P}_j} \alpha_{\mathscr{p}}, \tag{3.30}$$

where $\mathcal{P}_j$ is the set of all paths leading to the terminal node representing value j. Unfortunately, even the improved approach does not scale well since the number of possible paths is still considerable – it may depend on the number of variables exponentially.

Fortunately, it is possible to count the number of *satisfying* input vectors – which, in our case, is equivalent to the number of state vectors – of an integer function represented by MDD using the *satisfy-count* algorithm described in section 3.2.4.

The solution utilizing the *satisfy-count* algorithm is efficient with respect to the number of nodes – the complexity of the algorithm is $O(\mathcal{s})$ where $\mathcal{s}$ is the number of nodes in the diagram. However, it has a technical limitation. The number $\alpha_{\phi,j}$ can be very large even for tens of variables. The problem is that many programming languages are limited by the finite precision of their numeric types – typically 64-bit integers. The problem can be addressed by using a library for multiple precision arithmetic such as *GMP* [87]. Nevertheless, the computation for larger functions – containing hundreds or thousands of variables – would involve computation with huge numbers and therefore could be time-consuming.

This problem can be partially addressed in the case of BSS analysis. A modification of the *satisfy-count* algorithm exists called *satisfy-count-ln*[4], which – as the name suggests – calculates the logarithm of the number of satisfying input vectors i.e., the number $\log_2 \alpha_{\phi,j}$. Since the algorithm works with logarithms, it is not susceptible to integer overflows. Knowing the logarithm $\log_2 \alpha_{\phi,j}$, we can subsequently calculate the state frequency by rewriting the definition (3.26) in terms of logarithms in the following way:

$$\begin{aligned} Fr^{=j} &= \frac{\alpha_{\phi,j}}{\alpha_\phi} \\ &= \frac{2^{\log_2(\alpha_{\phi,j})}}{2^{\log_2(\alpha_\phi)}} \\ &= 2^{\log_2(\alpha_{\phi,j})-\log_2(\alpha_\phi)} \\ &= 2^{\log_2(\alpha_{\phi,j})-\log_2(2^n)} \\ &= 2^{\log_2(\alpha_{\phi,j})-n}. \end{aligned} \tag{3.31}$$

[4] Implemented in BuDDy BDD package [68]

The above solution does not scale to MSS – especially nonhomogeneous MSS. One of the reasons is that the denominator α_ϕ is a product (3.27) that cannot be simplified to single exponentiation and therefore cannot be further simplified using logarithm. The key to efficient calculation is to avoid the computation of the number of state vectors altogether. We propose a simple general algorithm that can be used to analyze nonhomogeneous MMS, and consequently homogeneous MSS and BSS. The algorithm follows the general structure of diagram manipulation (Alg. 3.6). Its pseudocode is presented in Alg. 3.7 and Alg. 3.8. The algorithm itself is a simplified version of the probabilistic algorithms described further in section 5.1.

```
procedure StateFrequency(diagram, j)
    root ← Root(diagram)
    frequency ← StateFrequencyStep(root, j)
    return frequency
end procedure
```

Alg. 3.7 Entry point of the state-frequency algorithm

```
procedure StateFrequencyStep(node, j)
    if IsTerminal(node) ∧ Value(node) = j then
        return 1.0
    end if
    if IsTerminal(node) ∧ Value(node) ≠ j then
        return 0.0
    end if
    if Contains(memo, node) then
        return Lookup(memo, node)
    end if
    frequency ← 0.0
    i ← Index(node)
    for k = 0 to $m_i$ do
        son ← Son(node, k)
        sonFrequency ← StateFrequencyStep(son)
        frequency ← frequency + sonFrequency ∗ (1 / $m_i$)
    end for
    Put(memo, node, frequency)
    return frequency
end procedure
```

Alg. 3.8 Recursive step of the state-frequency algorithm

Our algorithm is general and therefore there is no need to use the specialized version that utilizes *satisfy-count-ln* for BSS. Furthermore, our algorithm uses only addition and multiplication whereas the other one uses exponentiation and logarithms extensively. This suggests that our algorithm should perform better in the case of BDD. Nevertheless, an experimental comparison of the two algorithms is needed to confirm this assumption.

3.3.3 Probabilistic Analysis

In section 1.5 we described probabilistic analysis as a more precise way to analyze a system since, in addition to the system topology, it also considers the reliability of its components. Therefore, the input of algorithms for the probabilistic analysis is the diagram representing the structure function and the component state probabilities (1.14).

One of the fundamental tasks of probabilistic analysis is the calculation of system state probabilities. This task involves the evaluation of the probability that the structure function $\phi(\boldsymbol{x})$ evaluates to value j. System state probability is closely tied to system availability (1.19) and unavailability (1.20) – one can be computed in terms of the other. Calculation of various importance measures also involves the evaluation of the probability that a derivative of the structure function evaluates to 1. Hence, the fundamental algorithm for the probabilistic analysis is the algorithm that calculates the probability that a function represented by MDD evaluates to value j given component state probabilities (1.14).

The probability that MDD evaluates to value j agrees with the so-called Node Traversing Probability (NTP) [48] of the terminal node representing value j. Before we proceed with the calculation of NTP, we first need to consider *path probability*. Recall that a path is an alternating sequence of internal nodes and edges that lead to a terminal node. For the calculation, let us view the path as a sequence of pairs (i_l, k_l) where i_l is the index of variable associated with l^{th} internal node and k_l is the edge we chose in the l^{th} internal node. Assuming that the component state probabilities are independent we can calculate the path probability ρ of path $\wp$ as [48]:

$$\rho_{\wp} = \prod_{(i,k)\in\wp} p_{i,k}, \tag{3.32}$$

we calculate the NTP of a terminal node T_j as the sum of path probabilities leading to T_j as:

$$\text{NTP}(T_j) = \sum_{\wp}^{\mathcal{P}_j} \rho_{\wp}. \tag{3.33}$$

Definition (3.32) shows that we can associate component state probabilities with edges in the MDD. We refer to such MDD as *probabilistic MDD* (Fig. 3.22). Considering the implementation of the MDD, it would not be efficient to store the probabilities directly in the edges – the same information would be stored multiple times. Therefore, the implementations typically store the probabilities in a matrix $\mathbb{P}_{n,m}$. Nevertheless, the visualization of probabilistic MDD is good for understanding probabilistic calculations.

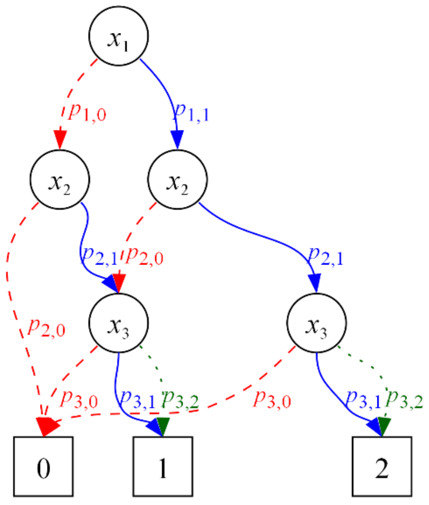


Fig. 3.22 Probabilistic decision diagram with component state probabilities attached to edges

In section 3.3.2 we have shown that calculation involving enumeration of all possible paths is not effective. Thus, an efficient evaluation of $\mathrm{NTP}(T_j)$ needs a more sophisticated approach. In section 5.1, we describe two principal approaches to the calculation along with their use cases and experimental comparison.

Furthermore, so far in this section we only described the time-independent version of the probabilistic analysis. Therefore, in section 5.2, we discuss how to adjust existing algorithms for the time-independent analysis to use them with time-dependent probabilities.

3.3.4 Logic Derivatives

Logic derivative is an essential tool for the calculation of different reliability characteristics – mostly for the calculation of various importance measures as described in section 2.3. Therefore, the calculation of the logic derivative of the function represented by MDD is another fundamental task of reliability analysis.

A straightforward approach to the calculation is to follow the definition of the derivative. However, different types of derivatives exist (introduced in section 2.2.2) – each with a slightly different definition. For the moment, let us consider the directional logic derivative (2.12). Alg. 3.9 contains pseudocode for the calculation of the derivative using diagram manipulation algorithms introduced in section 3.2.4. Notice that the code simply calls algorithms corresponding to operations used in the definition. The notation of the second argument of the TRANSFORM algorithm $(= j)$ denotes a single-parameter function that returns true if the parameter is equal to j and false otherwise. This is known as partial function application [88] – the original function being the two-parameter equals $(=)$ function. Also, notice that we can merge the diagrams using logical conjunction $\wedge$ since the call to the transform algorithm ensures that both diagrams have Boolean-valued output. Fig. 3.24 shows a specific example of the calculation of the derivative using the algorithm Alg. 3.9.

```
procedure DPLD(diagram, i, j, h, s, r)
    before ← COFACTOR(diagram, i, s)
    after ← COFACTOR(diagram, i, r)
    before’ ← TRANSFORM(before, (= j))
    after’ ← TRANSFORM(after, (= h))
    result ← APPLY(before’, after’, ∧)
    return result
end procedure
```

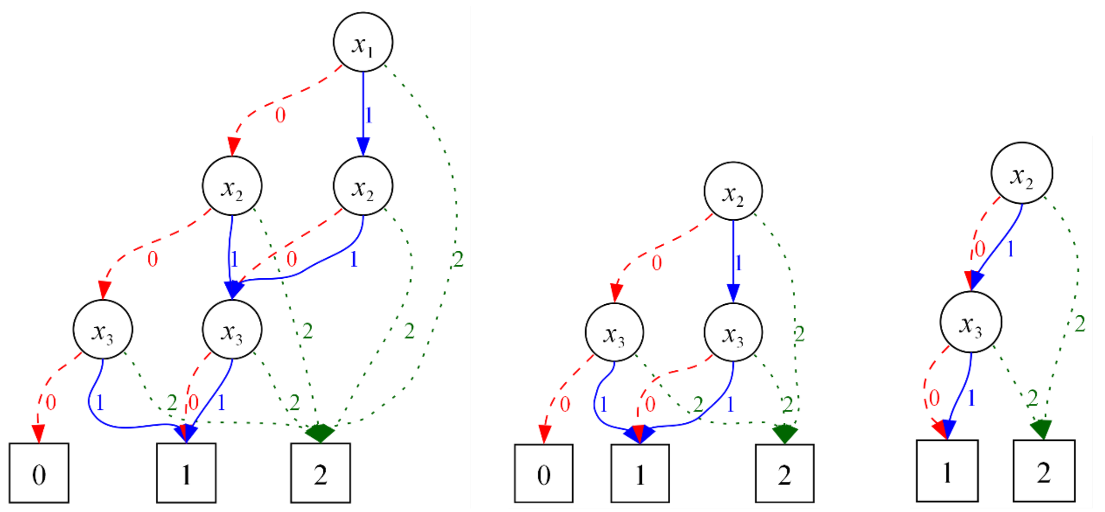
Alg. 3.9 Calculation of directional logic derivative

Let us consider one more example using the same approach – the calculation of the integrated directional logic derivative of type II (2.19). The pseudocode for this calculation can be found in Alg. 3.10. Its structure is very similar to Alg. 3.9, the only difference is that it does not involve any transformation of the diagrams and that it uses the greater than $>$ operator to merge the diagrams.

```
procedure IDPLDTYPEII(diagram, i, s, r)
    before ← COFACTOR(diagram, i, s)
    after ← COFACTOR(diagram, i, r)
    result ← APPLY(before, after, >)
    return result
end procedure
```

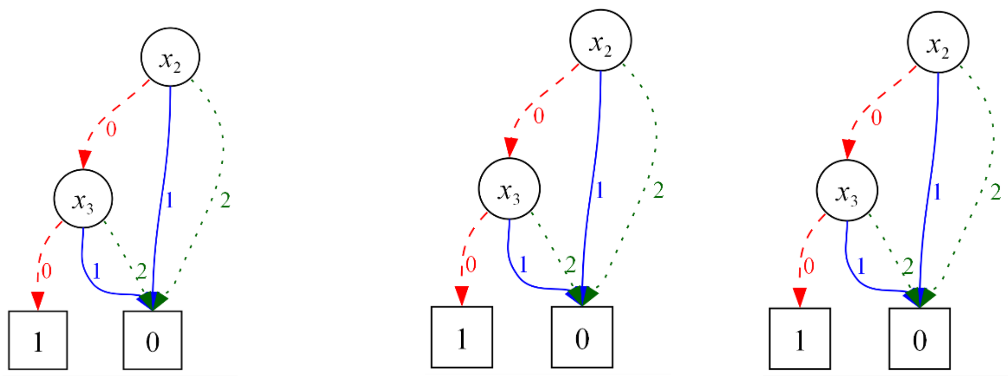
Alg. 3.10 Calculation of integrated directional logic derivative of type II

Fig. 3.23 Fig. 3.24 and show an example of the calculation of DPLD of type I using the approach described by Alg. 3.9. We could use a similar approach to calculate the other derivatives introduced in section 2.2.2. The derivatives can be used to evaluate various system characteristics such as MCVs [89] or IMs such as SI [41]. Furthermore, in conjunction with component state probabilities, we can evaluate more IMs such as BI [41] and use further transformations of the derivative to calculate FVI [43], [44]. Consequently, we can see that decision diagrams and logic derivatives provide a comprehensive framework for the reliability analysis of complex systems.



$\phi(\boldsymbol{x})$ *diagram* $\phi(0_1, \boldsymbol{x})$ *before* $\phi(1_1, \boldsymbol{x})$ *after*

Fig. 3.23 MDDs representing structure function and intermediate diagrams used in the calculation of directional logic derivative



$\phi(0_1, \boldsymbol{x}) = 0$ *before'* $\phi(1_1, \boldsymbol{x}) = 1$ *after'* $\dfrac{\partial\phi(0 \to 1)}{\partial x_1(0 \to 1)}$ *result*

Fig. 3.24 MDDs representing intermediate diagrams and resulting diagram representing the directional logic derivative

We have presented a basic approach to the calculation of logic derivatives. As long as we have an MDD of reasonable size representing the structure function, the calculation of the derivative is also reasonably efficient. The most expensive step of the calculation is the call to the *apply* algorithm (refer to section 3.2.4). However, even with the two presented examples, we may have noticed that the calculation is almost identical. Hence, in section 4.5 we present a single universal algorithm for the calculation of any derivative along with an experimental comparison with the basic approach.

3.4 MDD-Related Tasks Open for Investigation

In section 3.3, we established the principal steps of the reliability analysis using decision diagrams. The steps can be summarized as follows:

- construction of the structure function;
- creation of decision diagrams;
- adjustments and transformations of the diagrams;
- evaluation of the diagrams;
- and interpretation of the results.

The listed problems involve several challenges in the context of analysis of complex systems. Consequently, we have identified multiple research problems that address the challenges. Some of the problems involve experimental comparison of existing algorithms in different use cases whilst others require an introduction of new algorithms or enhancements of existing algorithms. Specifically, we have identified the following research problems:

- generating random decision diagrams representing structure functions – required for exhaustive experimental comparisons;
- order of diagram merging and its influence on the speed of diagram creation (section 3.3.1);
- different ways of representation of a structure function of a series-parallel system and their influence on the size of the diagram (section 3.3.1);
- experimental comparison of our algorithm for the calculation of state frequency with alternative approaches for BDDs (section 3.3.2);
- introduction of a new universal algorithm for the calculation of logic derivatives and experimental comparison with existing approaches (section 3.3.4);
- experimental comparison of existing algorithms for the probabilistic evaluation of decision diagrams (section 3.3.3);
- adjustment of existing algorithms for probabilistic evaluation of decision diagrams with time-dependent probabilities (section 3.3.3).

4 Efficient Diagram Creation and Manipulation

The structure function is an integral part of the reliability analysis process. Considering that complex systems consist of numerous components it is essential to represent them efficiently. In this thesis, we focus on decision diagrams, which have proven to be a suitable structure for the task [44], [81], [90]. However, even though the diagrams are efficient, their size can be considerably high for large systems. Therefore, it is important to develop algorithms and approaches that can speed up diagram creation and manipulation.

4.1 Generating Random Diagrams

Chapter 4 and Chapter 5 deal with efficient diagram manipulation, which involves a considerable number of experimental comparisons. A comparison should be done, ideally, on diagrams with different structures and sizes. Therefore, in this section, we describe the methods we use for the generation of random decision diagrams.

4.1.1 Min-Max Expressions

The first approach is based on SoP expressions, specifically, on their generalized form where the logical conjunction is generalized using the min function and logical disjunction using the max function. The reason for this choice of generalization is that it is commonly used in the description of series-parallel systems (section 1.3.1). Another reason is that it is easy to represent and generate such expressions. For example, let us consider the following expression:

$$f(\boldsymbol{x}) = \max(\min(x_1, x_2, x_3), \min(x_1, x_3, x_4), \min(x_2, x_3, x_4)). \tag{4.1}$$

We can represent the expression conveniently using the following list of lists of integers:

$$[[1,2,3], [1,3,4], [2,3,4]].$$

Furthermore, we can also generate such a list conveniently by generating random integers from the range $[1,2, \dots, n)$ where n is the number of variables. The algorithm for the generation has the following parameters:

- number of terms – the number of nested lists;
- size of a term – the number of variables in a single term;
- n – the number of variables;
- type of the function:
    - Boolean,

- MVL – requires additional parameter m,
- or integer – requires additional parameters m and m_i for $i = 1,2,\dots,n$.

Subsequent creation of a diagram from such an expression is straightforward. We start by creating a diagram representing each nested list using the min operation and then we proceed with the merge of the diagrams using the max operation. This process is also known as *fold*, which we describe in section 4.2.1.2.

4.1.2 Series-Parallel Trees

The min-max approach is suitable for generating general MDDs. The average size of the generated MDD can be influenced by adjusting values of the parameters e.g., the number of terms or size of a term. A possible drawback of the approach is that generated MDDs do not correspond to the structure function of specific system types. Therefore, the second approach that we use – which we call the *series-parallel trees* approach – aims to generate MDDs representing structure functions of series-parallel systems (section 1.3.2).

To generate such MDD, we use the same process as with the min-max expression. We start by generating a description of the system, which we subsequently transform into MDD. The description that we chose for the series-parallel system is the Abstract Syntax Tree (AST), which we briefly described in section 2.4.1. As an example, let us consider the series-parallel system depicted in Fig. 1.4 and let us assume that we use the min and max functions to describe series and parallel connections respectively. In Fig. 4.1, we can see AST representing the system.

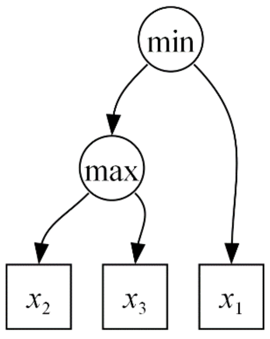


Fig. 4.1 AST representing series-parallel system depicted in Fig. 1.4

Creation of the diagram from an AST can be done in a simple post-order traversal of the tree. In terminal nodes of the tree, we create diagrams representing given variables, and in internal nodes, we merge diagrams created in the traversal of the sons using the *apply* algorithm with min or max operation. We also use a recursive procedure to generate the

AST itself. The procedure has a single parameter n – the number of variables that the tree (subsystem) should contain. The terminating case of the recursion is when the value of the parameter is 1, in which case we create a diagram representing the variable with the next index from a sequence, which must be shared amongst all recursive calls. In the non-terminating case, we:

- split the number of variables n in half (or in any other ratio),
- randomly choose either min or max operation,
- generate sons of the new node using a recursive call,
- and finally return a new internal node.

In Alg. 4.1 we can see a pseudocode of the above procedure.

```
procedure GenerateRandomAST($n$)
    if $n = 1$ then
        return Node(NextIndex())
    end if
    $op \leftarrow$ SelectRandomElement({min, max})
    $leftSize \leftarrow n/2$
    $rightSize \leftarrow n - leftSize$
    $left \leftarrow$ GenerateRandomAST($leftSize$)
    $right \leftarrow$ GenerateRandomAST($rightSize$)
    $node \leftarrow$ Node($op$, $left$, $right$)
    return $node$
end procedure
```

Alg. 4.1 Recursive procedure for the generation of random AST representing a series-parallel system

4.2 Improvement of Dynamic Creation

4.2.1 Order of Evaluation

4.2.1.1 Left Fold and Tree Fold

The creation of decision diagrams can be a complicated process that can be approached in several ways – some of which we describe in section 3.2.2. Each approach has its use case, but the dynamic creation using the *apply* algorithm plays a pivotal role. During the creation, we often encounter a situation when we need to merge several diagrams (sequence of diagrams) using an associative operation $\oplus$. A typical example of this situation is the creation of a diagram representing SoP expression or min-max expression since logical conjunction, logical disjunction, min, and max are associative all operations. As we described in section 4.1.1, to create a diagram for such an expression, we start by directly creating diagrams representing products, and then we proceed to merge them using the sum operation.

The associativity of the merging operation allows us to join diagrams in numerous ways, using different orders of evaluations, while still achieving the same result. It is interesting to consider two orders that we encounter in other areas such as functional programming [88]. The first intuitive order is called *left fold* since it simply merges the sequence of diagrams from left to right. In Fig. 4.2 we can see a tree illustrating the *left fold* order of evaluation where the $\oplus$ nodes represent the merger of diagrams using associative operation and the house-shaped nodes represent the initial sequence of diagrams.

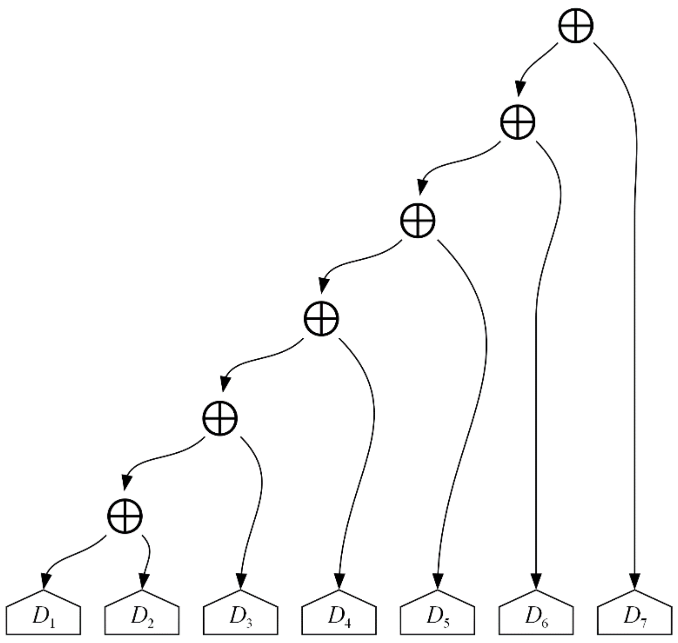


Fig. 4.2 Left fold order of evaluation

The second order of evaluation is called *tree fold* and works in a slightly less intuitive yet elegant way. Compared to the *left fold*, which works sequentially, the *tree fold* operates hierarchically by incrementally merging pairs of neighboring diagrams until there is only one resulting diagram left. In Fig. 4.3 we can see the *tree fold* order of evaluation using the same notation as in Fig. 4.2.

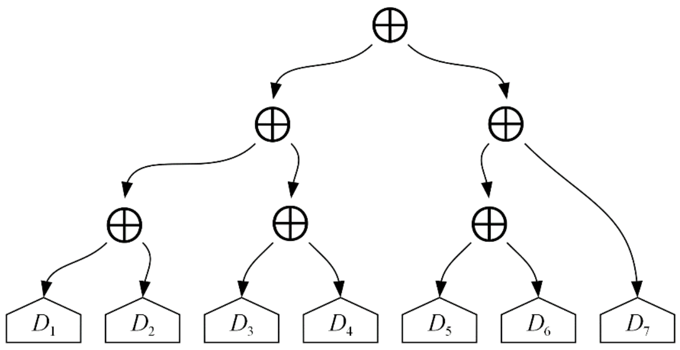


Fig. 4.3 Tree fold order of evaluation

Notice that the number of diagram mergers (the number of $\oplus$ nodes) is the same for both approaches. However, an interesting question is whether different orders of evaluation can influence the speed of diagram creation and whether we can identify properties of the diagrams that would allow us to pick favorable orders for a specific use case.

4.2.1.2 Fold Comparison

To investigate the influence of the order of evaluation we performed an experimental comparison of the two *folds* in the creation of BDDs representing SoP expressions. We used functions defined in PLA format [9], which is a compressed form of a truth table that can be easily read as a SoP expression and subsequently transformed into a decision diagram.

4.2.1.2.1 Boolean Functions Representing Adders

In the first experiment, we examined functions representing output bits of bit adders [91] measuring the time needed to create diagrams representing all outputs of the adder circuits. In Tab. 4.1, we can see the properties of the functions examined in the experiment. The number of terms sets the upper bound on the number of terms in the SoP that we merged to obtain the resulting diagram (some of the terms were skipped in the creation of some functions because the function did not depend on them). The number of functions agrees with the number of output bits of the adder and therefore with the number of diagrams created for the given file. Finally, in Tab. 4.2 we can see a summary of the results that we presented in the paper [92]. The results clearly show that for each file the *tree fold* approach was the more efficient in time required to create diagrams for each output function defined in the file.

Tab. 4.1 Properties of the functions used in the experiment

File name	Number of terms	Number of variables	Number of functions
10-adder_col	10,191	21	11
11-adder_col	20,427	23	12
12-adder_col	40,911	25	13
13-adder_col	81,867	27	14
14-adder_col	163,783	29	15

Tab. 4.2 Average time in milliseconds needed to create BDDs representing outputs of the adder

File name	Left fold [ms]	Tree fold [ms]
10-adder-col	163	71
11-adder-col	477	180
12-adder-col	1,828	448
13-adder-col	5,342	1,084
14-adder-col	16,579	2,753

The PLA files from the benchmark that we used in the experiment stored the products (rows of the compressed truth table) in a specific configuration, which might influence the result of the experiments since it might not represent a general case. Due to that, we repeated the experiment but before diagram creation, we randomly shuffled the rows of the file. Tab. 4.3 presents the results of the second version of the experiment. The first observation is that the total time needed for the diagram creation is considerably higher in both cases. Moreover, the second and more important observation is that the *tree fold* approach was significantly slower, which is the opposite of the results of the first experiment.

Tab. 4.3 Average time in milliseconds needed to create BDDs representing outputs of the adder with randomly shuffled rows

File name	Left fold [ms]	Tree fold [ms]
10-adder_col	522	789
11-adder_col	2,075	2,975
12-adder_col	7,218	13,031
13-adder_col	27,512	51,063
14-adder_col	94,292	230,151

4.2.1.2.2 Various Boolean Functions

The results that we obtained from the first experiment are restricted to the specific type of function representing bit adders. To obtain more general results we repeated the experiment using a different – more representative set of 39 functions from the IWLS'93 benchmark set [93]. Tab. 4.4 presents a summary of the results that we presented in the paper [94]. The value of δ in the first column of the table indicates the tolerance used to determine which version of the fold is considered faster. The fold strategy is considered faster if the ratio of the time needed to create the diagrams using *tree fold* (numerator) and *left fold* (denominator) is less than $1 - \delta$. Obviously, with decreasing value of δ (with decreasing tolerance), the number of cases when the *left fold* was faster and when the *tree fold* was faster equalizes. The results show that in a more general set of functions, either of the two folds can be faster depending on the specific function.

Tab. 4.4 Number of functions in the benchmark in which speed of *left fold* and *tree fold*-based merging are different with respect to value δ

$\boldsymbol{\delta}$	**Faster**	
	Left fold	**Tree fold**
0.10	3	10
0.05	10	17
0.01	17	19
0.00	19	20

Both experiments showed that the choice of the folding strategy can have a significant impact on the speed of diagram creation. Unfortunately, the results imply that neither one of the examined folding strategies is more efficient in general, though the first experiment implies that there exists a specific configuration that favors the *tree fold* approach. Therefore, in general, it is advantageous for decision diagram libraries to implement both folding strategies so that the user can test and choose the strategy that is more suitable for his use case.

4.2.2 Extended Apply

The *extended apply* algorithm that we introduced in section 3.2.3 provides a more convenient way of the creation of decision diagrams that represent d-ary operations. To examine the practical performance of the algorithm, we performed an experimental comparison [95] with the basic version of the *apply* algorithm.

In the experiment, we generated random ASTs of different sizes using an approach similar to the one described in section 4.1.2, with a difference that the generated trees were d-way trees (each internal node has d outgoing edges) with $d = 2,3,4,5$. The size of the tree was given by the parameter n_{max}, which set the number of leaf nodes of the AST. Then, we transformed the AST into BDD using the *extended apply algorithm* as shown in Tab. 4.5.

The experiments measured the average time in milliseconds (obtained from 1,000 replications) required to transform randomly generated AST into BDD and the average required number of steps of the algorithm. The results of the comparison are presented in Tab. 4.6 and Tab. 4.7.

Tab. 4.5 Usage of the extended apply algorithm with different arities in the creation of BDD from an AST (the last parameter is omitted for clarity)

$\boldsymbol{d}$	**Extended apply calls**
2	$\text{APPLY}(\text{APPLY}(\text{APPLY}(\text{APPLY}(D_1, D_2), D_3), D_4), D_5)$
3	$\text{APPLY}(\text{APPLY}(D_1, D_2, D_3), D_4, D_5)$
4	$\text{APPLY}(\text{APPLY}(D_1, D_2, D_3, D_4), D_5)$
5	$\text{APPLY}(D_1, D_2, D_3, D_4, D_5)$

Tab. 4.6 The average time in milliseconds requires to create BDD from AST

$\boldsymbol{n_{max}}$	$\boldsymbol{d}$			
	2	**3**	**4**	**5**
20,000	79	78	94	113
40,000	177	176	209	252
60,000	280	278	333	401
80,000	384	383	457	550
100,000	500	495	592	712

Tab. 4.7 The average number of steps of the extended apply algorithm

$\boldsymbol{n_{max}}$	$\boldsymbol{d}$			
	2	**3**	**4**	**5**
20,000	500,347	410,656	498,364	555,554
40,000	1,055,979	864,883	1,053,838	1,172,272
60,000	1,637,588	1,339,495	1,643,756	1,827,908
80,000	2,227,090	1,818,588	2,232,094	2,479,476
100,000	2,831,173	2,311,067	2,836,996	3,149,147

Overall, the results of our comparison show that the basic version of the *apply* algorithm performs better compared to the extended versions – and thus we do not achieve a significant speedup by using the extended algorithm. On the other hand, the version with arity 3 proved to be equally fast and more efficient in terms of the number of steps of the algorithm, which suggests that there may be use cases where the extended versions are more appropriate. Finally, we consider one of the important benefits of extended apply to be the convenience of using the algorithm when creating diagrams for functions that are d-ary in their nature.

4.3 Representation of Series-parallel Systems

Series-parallel systems are one of the system types that we consider complex when they consist of a high number of components. Their nature also allows us to efficiently generate systems with random topologies (section 4.1.2). Therefore, we chose this topology to compare different approaches to structure function representation.

4.3.1 Comparison of Single and Series of Diagrams

In the experimental examination presented in our paper [96], we generated a random series-parallel MSS and created a single and series of diagrams representing the structure function of the system. We compared the number of unique nodes needed to represent the structure function using both approaches. Tab. 4.8 presents a summary of the results, which clearly show that the series approach is more efficient considering the number of unique nodes. The number of nodes is one of the key properties of a decision diagram because it defines the complexity of many algorithms that operate on the diagrams [11]. Therefore, the possibility of representing series-parallel systems more compactly using the series of diagrams has a positive impact on our ability to analyze complex series-parallel systems.

Tab. 4.8 Average number of nodes in a single MDD and in a series of MDDs depending on the number of system components (n) in case of homogeneous series-parallel 3, 4, and 5 state MSS

n	**Single MDD**			**Series of MDDs**		
	3	4	5	3	4	5
500	1,995	5,138	10,756	1,002	1,502	2,002
1,000	4,232	11,436	24,926	2,002	3,002	4,002
1,500	6,562	18,210	40,591	3,002	4,502	6,002
2,000	8,959	25,346	57,424	4,002	6,002	8,002
2,500	11,406	32,749	75,128	5,002	7,502	10,002

4.3.2 Influence of the Order of Variables

One of the limitations of the presented experiment is that we only considered a single order of variables in the diagram – the default order of variables. However, as we showed in section 3.1.5, the order of variables can significantly influence the number of nodes. Thus, to examine whether the series approach is more efficient even with an arbitrary order of variables we performed a second experiment. The new experiment had a similar setup as the previous experiment but in addition to generating a random topology of the system, we generated and used random order of variables. Also, since some order of variables can result in an impractical number of nodes, we significantly reduce the number of components of the generated system. In table Tab. 4.9 we can see the results of the new experiment presented in our paper [97]. We can see that despite a significantly lower number of components the number of nodes in the diagrams (especially single diagrams) is considerably higher. This suggests that the default order of variables that we used in the first experiment is a reasonable and efficient choice for series-parallel systems. Most importantly, the data show that the series approach is more efficient even in situations when we use a random order of variables.

Tab. 4.9 Average number of nodes in a single MDD and in a series of MDDs depending on the number of system components (n) in case of homogeneous series-parallel 3, 4, and 5 state MSS

n	**Single MDD**			**Series of MDDs**		
	3	4	5	3	4	5
10	61	134	268	40	58	78
20	797	3,495	11,782	227	335	440
30	9,381	84,858	545,158	1,096	1,620	2,194

The results of the experiments offer two interesting conclusions. The first conclusion is that the individual description of systems states of MSS is considerably more effective than the description of the entire system. The second conclusion is that the default order of variables is a reasonable choice for decision diagrams representing structure functions of series-parallel systems.

4.4 System State Frequency Evaluation

The calculation of system state frequency involves multiple challenges that we need to deal with for it to be efficient. We identified three principal approaches to the calculation, which we discussed in detail in section 3.3.2. At the end of the section, we presented a general algorithm (Alg. 3.7) for the calculation that can be used for BSS as well as MSS. However, in the case of BSS, there exists an alternative approach that utilizes logarithms to avoid integer overflow. We assume that our general algorithm should perform better. To confirm the assumption, we compared the three approaches experimentally.

In the experiment, we generated 200 random BDDs using the min-max approach (section 4.1.1) for different numbers of variables (n). Subsequently, we computed the system state frequency using all three approaches. In Tab. 4.10, we can see the average time in microseconds required to calculate the state frequency using the three approaches. Notice that we used the GMP multiple precision arithmetic library.

Tab. 4.10 Average time in microseconds required to calculate the state frequency using different approaches

n	**Satisfy-count [μs]**	**Satisfy-count-ln [μs]**	**Our [μs]**
10	12	14	10
30	1,387	1,823	1,337
60	59,157	69,047	57,107
80	[a] 471,942	191,991	161,950
90	[a] 788,309	329,563	287,166
100	[a] 1,057,991	470,303	407,762

[a] Using the GMP integers

The results of the experiment confirmed our assumption that our algorithm performs better than the approach that utilizes logarithms. It also confirmed our assumption that even though it is possible to use the basic approach based on the *satisfy-count* algorithm – which performs comparably for $n < 63$ – the calculations involving multiple precision integers are considerably slower. Therefore, we conclude that it is better to use our algorithm even for the special case of BSS.

4.5 Efficient Calculation of Logic Derivatives

4.5.1 Parametrized Procedure

Calculation of logic derivatives is an important step of the reliability analysis process. In section 3.3.4 we described a possible approach to the calculation that utilizes general diagram manipulation algorithms and can be used to calculate all types of derivatives. However, one of the drawbacks of the presented approach is that it requires a separate procedure for each type of derivative. As an example, we presented such procedures in Alg. 3.9 and Alg. 3.10.

The first step in the improvement of the approach is the observation that the examples – and procedures for the calculation of other types of derivatives as well – have almost identical structures and, therefore, can be parameterized. Thus, the task is to identify the parameters. Except for the IDPLD of type II, the derivatives differ only in the transformation that they use on the cofactors. Hence, the first pair of parameters are two transformation functions – γ_{left} and γ_{right}. Type II is the only one that uses other operations than $\wedge$ in the final apply call. Therefore, the operation also needs to be a parameter. Finally, let us notice that the change in the value of the variable does not require parametrization since it is the same for all types of considered derivatives. Considering all the parameters, we present the pseudocode of the parametrized procedure in Alg. 4.2.

```
procedure ParametrizedDPLD($diagram$, $s$, $r$, $\gamma_{left}$, $\gamma_{left}$, $\odot$)
    $before$ ← Cofactor($diagram$, $i$, $s$)
    $after$ ← Cofactor($diagram$, $i$, $r$)
    $before'$ ← Transform($before$, $\gamma_{left}$)
    $after'$ ← Transform($after$, $\gamma_{right}$)
    $result$ ← Apply($before'$, $after'$, $\odot$)
    return $result$
end procedure
```

Alg. 4.2 Parametrized procedure for the calculation of any (I)DPLD

The algorithm presented in Alg. 4.2 can be used to calculate all types of derivatives by providing appropriate values of the parameters. In table Tab. 4.11 we present parameters for the calculation of all derivative types described in section 2.2.2. The notation $(= j)$ follows the syntax of partial function application [88] used in some programming languages. The presented example $(= j)$ denotes an anonymous unary function that returns true if and only if its argument equals the value j. In the special case of IDPLD of type II we use the identity function that returns its argument unchanged.

Tab. 4.11 Parameters of the parametrized procedure for the calculation of any (I)DPLD

Derivative	**Left transform (γ_{left})**	**Right transform (γ_{right})**	**Apply operation ($\odot$)**
DPLD	$(= j)$	$(= h)$	$(\wedge)$
IDPLD Type I	$(= j), (< j)$	$(< j), (= j)$	$(\wedge)$
IDPLD Type II	$\lambda a. a$ [a]	$\lambda a. a$ [a]	$(<), (>)$
IDPLD Type III	$(< j), (\geq j)$	$(\geq j), (< j)$	$(\wedge)$

[b] The identity function

4.5.2 Specialized (I)DPLD Calculation Algorithm

4.5.2.1 Introduction of the Algorithm

The procedure presented in Alg. 4.2 is reasonably efficient – the most expensive step is the final *apply* call. Therefore, if we can represent the structure function with a diagram of reasonable size, we can analyze it using logic derivatives. On the one hand, the advantage of the presented procedure is that it uses a general diagram manipulation algorithm and, thus, can be used with general diagram manipulation libraries. On the other hand, the calculation of the derivatives has certain specifics that the general approach cannot exploit. Therefore, we present an algorithm designed specifically for the calculation of logic derivatives.

Let us consider the transformed cofactors that enter the final *apply* call. The two diagrams originate from the same diagram and, therefore, their structure is quite similar. Moreover, we know exactly how they differ – the difference is only in the edge we used to "skip" a node representing the variable x_i. This allows us to avoid "materializing" the intermediate results (the cofactors). Instead, we can use only a *view* of the original diagram, which uses a modified version of son access – the function GETSON presented in Alg. 4.3. Consequently, we can skip the calculation of the cofactors and instead use the views (with appropriate parameters) as the input of the apply call.

Another intermediate result that we would like to avoid is the calculation of the transformed diagrams. Fortunately, we can use the same approach with a view of the diagram as we did in the case of the cofactor. If we ignore type II, we notice that the role of the transformations is to transform the cofactors into pseudo-logic functions so they can be merged using *apply* with ∧ operation. Therefore, the solution is to use a custom operation for the *apply* call – let us denote it using the letter Λ. The operation is a function of two parameters of the form (3.12). The functions aim to first transform the parameters and then return true if the values describe the desired change and false otherwise. The Λ operation allows us to define the derivative in the following universal way:

$$\frac{\partial f(\Lambda)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{if } \Lambda\big(f(s_i, \boldsymbol{x}), f(r_i, \boldsymbol{x})\big) \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

Furthermore, from the practical point of view, it is even better if the Λ operation encodes the values $\{\text{true}, \text{false}\}$ with integers $\{1,0\}$. This allows us to write the definition without the if condition in the following form:

$$\frac{\partial f(\Lambda)}{\partial x_i(s \to r)} = \Lambda\big(f(s_i, \boldsymbol{x}), f(r_i, \boldsymbol{x})\big). \tag{4.3}$$

By combining the above-described ideas, we derived a universal algorithm for the calculation of arbitrary (I)DPLD within a single "apply-like" algorithm (without intermediate results). The algorithm uses an auxiliary function GETSON (Alg. 4.3) to access the son of a node. This function performs the "cofactoring" by skipping nodes representing the "derived by" variable.

```
procedure GetSon(node, k, i, value)
    son ← Son(node, k)
    if IsInternal(son) ∧ Index(son) = i then
        return Son(son, value)
    else
        return son
    end if
end procedure
```

Alg. 4.3 Helper function used in the step of the universal DPLD algorithm

The entry point of the algorithm is presented in Alg. 4.4. It handles the special case when we derive by the variable that is in the root of the diagram and, mainly, it calls the recursive step of the algorithm. The recursive step has a structure similar to the step of the *apply* algorithm. Its pseudocode is presented in Alg. A.5.

Finally, to be able to use the new algorithm, we need to define the Λ operation corresponding to all types of considered derivatives. In Tab. 4.12, we present the definitions for the calculation of all derivative types described in section 2.2.2 in the notation of lambda calculus. The first part of each expression $\lambda a. \lambda b.$ defines two parameters of the function named a and b. The expression following the last dot defines the value of the function for given values of parameters.

```
procedure UniversalDPLD(diagram, i, s, r, Λ)
    oldRoot ← Root(diagram)
    if IsInternal(oldRoot) ∧ Index(oldRoot) = i then
        left ← Son(oldRoot, s)
        right ← Son(oldRoot, r)
    else
        left ← oldRoot
        right ← oldRoot
    end if
    newRoot ← UniversalDPLDStep(i, s, r, Λ, left, right)
    return MDD(newRoot)
end procedure
```

Alg. 4.4 Entry point of the universal DPLD algorithm

```
procedure UniversalDPLDStep($i$, $s$, $r$, $\Lambda$, $left$, $right$)
    if Contains($memo$, ($left$, $right$)) then
        return Lookup($memo$, ($left$, $right$))
    end if
    if IsTerminal($left$) $\wedge$ IsTerminal($right$) then
        $node \leftarrow$ MakeTerminalNode($\Lambda$(Value($left$), Value($right$)))
    else
        $i_{left} \leftarrow$ Index($left$)
        $i_{right} \leftarrow$ Index($right$)
        $i_{new} \leftarrow \min(i_{left}, i_{right})$
        sons $\leftarrow$ MakeTuple($m_{inew}$)
        for $k = 0$ to $m_{inew}$ do
            if $i_{left} = i_{new}$ then
                $lhs \leftarrow$ GetSon($left$, $k$, $i$, $s$)
            else
                $lhs \leftarrow left$
            end if
            if $i_{right} = i_{new}$ then
                $rhs \leftarrow$ GetSon($left$, $k$, $i$, $r$)
            else
                $rhs \leftarrow right$
            end if
            $sons[k] \leftarrow$ UniversalDPLDStep($i$, $s$, $r$, $\Lambda$, $lhs$, $rhs$)
        end for
        $node \leftarrow$ CreateInternalNode($i_{new}$, $sons$)
    end if
    Put($memo$, ($left$, $right$), $node$)
    return node
end procedure
```

Alg. 4.5 Recursive step of the universal DPLD algorithm

Notice that the variables j and h are not parameters of the Λ function – they need to be defined outside of the function and made available during the evaluation of the expression. Modern programming languages support this behavior in the form of lambda function variable captures, closures, function objects with member variables, or similar constructs.

Tab. 4.12 Functions used as the Λ parameter of our universal algorithm for the calculation of (I)DPLDs

Derivative	**Λ-operation**
DPLD	$\lambda a.\lambda b.(a = j) \wedge (b = h)$
IDPLD Type I	$\lambda a.\lambda b.(a = j) \wedge (b < j)$
IDPLD Type II	$\lambda a.\lambda b.(a < b)$
IDPLD Type III	$\lambda a.\lambda b.(a \geq j) \wedge (b < j)$

4.5.2.2 Experimental Comparison

Our algorithm performs the entire calculation within a single “apply-like” operation in contrast with the parametrized procedure (Alg. 4.2), which involves multiple diagrams traversing operations. Therefore, we assume that our algorithm should perform better when we consider the speed of the derivative calculation. The question is whether the assumption holds and if so, how big of a speedup our algorithm offers. The answer to the question should suggest whether it is worth implementing the new algorithm or whether using the simpler parametrized procedure provides comparable performance.

To answer the question, we compared the parametrized procedure and our approach presented in [98]. In the experiment, we generated random MDDs using the min-max approach (section 4.1.1) and measured the average time required to calculate IDPLD of type II and IDPLD of type III. We chose these two types because we assumed that the calculation times of other types are similar to those of type III. The average times for the different numbers of system states (m), number of variables (n) are presented in Tab. 4.13 and Tab. 4.14 with the relative performance (last column) of our algorithm.

Tab. 4.13 Average time in milliseconds required to compute IDPLD of type II for each variable using the parametrized procedure and using our algorithm

m	n	Node count	Parametrized procedure [ms]	Our algorithm [ms]	Our / Parametrized
2	32	129,448	2,069	1,533	0.7410
3	23	567,533	5,253	3,182	0.6058
4	20	1,641,815	16,399	9,740	0.5939
5	17	1,431,409	11,680	6,866	0.5879

Tab. 4.14 Average time in milliseconds required to compute IDPLD of type III for each variable using the parametrized procedure and using our algorithm

m	n	Node count	Parametrized procedure [ms]	Our algorithm [ms]	Our / Parametrized
2	32	128,322	3,570	1,538	0.43079
3	23	531,698	6,005	2,978	0.49597
4	20	1,591,344	18,540	9,625	0.51917
5	17	1,401,163	13,208	6,874	0.52042

The experimental comparison shows that our algorithm is roughly 50% faster than the general parametrized approach. Since the calculation of logic derivatives is one of the essential steps of reliability and importance analysis, our algorithm can provide a significant speedup to the process of complex system analysis.

5 Probabilistic Evaluation of Decision Diagrams

In section 3.3.3 we introduced the calculation of node traversing probability as an essential task of probabilistic system reliability analysis. Also, we presented practical challenges that arise in the computation. In this section, we address the challenges, starting with the description of algorithms for efficient NTP calculation and continuing with the description of the impact of time-dependent component state probabilities.

5.1 Calculation of Node Traversing Probabilities

Calculation of the NTP of a terminal node following the definition (3.33) would involve enumeration of all paths leading to a given node, which is computationally infeasible – as we established in section 3.3.2. Just like with the computation of the state frequency, computationally feasible algorithms use only a single traversal of the diagram. The literature recognizes two principal approaches, which are the *bottom-up* approach and the *top-down* approach.

5.1.1 Bottom-Up Approach

The *bottom-up* approach [10] is the simpler one of the two approaches. It calculates the sum of NTPs of selected terminal nodes by calculating the probability $\mathrm{Prob}(.)$ for each node using the following relation for internal node A:

$$\mathrm{Prob}(\mathcal{V}, A) = \sum_{k=0}^{m_{i_A}-1} \mathrm{Prob}(\mathcal{V}, A_k) * p_{i_A,k}, \tag{5.1}$$

and for terminal node B as:

$$\mathrm{Prob}(\mathcal{V}, B) = \begin{cases} 1.0, & \textsc{Value}(B) \in \mathcal{V} \\ 0.0, & \text{otherwise} \end{cases}, \tag{5.2}$$

where $\mathcal{V}$ is the set of values of selected terminal nodes. The sum of NTPs of all nodes representing values in $\mathcal{V}$ can be subsequently obtained using the following relation:

$$\sum_{B \in \mathcal{V}} \mathrm{NTP}(B) = \mathrm{Prob}(\mathcal{V}, root). \tag{5.3}$$

The recursive nature of the relation (5.1) directly translates to the recursive algorithm presented in Alg. A.7. A crucial aspect of the algorithm is that it visits each node just once – which is achieved using the memoization technique (section 3.2.4). Let us note that to calculate the $\mathrm{Prob}(\mathcal{V}, A)$, we first need to calculate $\mathrm{Prob}(\mathcal{V}, A_k)$ for $k = 0,1, \dots, m_{i_A} - 1$. This approach resembles the standard *post-order* traversal of a tree structure. Hence, we also refer to the algorithm as a *post-order* NTP calculation algorithm.

5.1.2 Top-Down Approach

The second approach is known in the literature as the *top-down* approach [99]. It calculates the NTP of each node using the following relation:

$$\mathrm{NTP}(A) = \sum_{(B,k)\, \in\, \mathcal{E}(A)} \mathrm{NTP}(B) * p_{i_B,k}, \tag{5.4}$$

where $\mathcal{E}(A)$ is a set of pairs of the form (k, B), which represents the set of all edges leading to node A – B being the source node and k denoting that A is k^{th} son of node B. The relation has the following special case for the root node:

$$\mathrm{NTP}(root) = 1.0. \tag{5.5}$$

Let us notice that the relations (5.2) and (5.4) are similar. The key difference is that in the case of the *top-down* approach (5.4), we first need to fully evaluate the probability in a node before we proceed with the evaluation of its sons – hence the name *top-down* approach. Also, let us notice the difference in the notation. In the *top-down* approach, we use the notation $\mathrm{NTP}(A)$, since the probability calculated in node A agrees with its NTP – this is one of the possible advantages of this approach. On the other hand, in the *bottom-up* approach, we denote the probability calculated in node A using the notation $\mathrm{Prob}(.\,, A)$ since the probability does not agree with its NTP.

In Alg. A.8 we can see the pseudocode of an algorithm implementing the *top-down* approach. This algorithm differs from other diagram-evaluating algorithms – it does not utilize recursion. The nature of the relation (5.4) requires that the diagram is processed using the breadth-first search (BFS) traversal, which is also known as *level-order* traversal in the context of tree-like structures. Hence, we also refer to the algorithm as a *level-order* NTP calculation algorithm.

Implementation of the BFS traversal requires an auxiliary data structure. This structure stores the nodes to be processed and is initialized in a way that it contains the root of the diagram. Furthermore, the structure must ensure that we first process all nodes with index i before we process any node with index $i + 1$, for $i = 1,2, \dots, n$ (assuming the default order of variables). Therefore, a suitable structure is a priority queue where the index of a variable associated with a node serves as the priority. Any implementation of the priority queue can be used; however, the increasing nature of priorities allows us to use a monotonic priority queue [100]. Specifically, we use a straightforward implementation of the bucket queue in the pseudocode (the stacks variable).

Furthermore, an additional constraint we need to consider is to ensure that each node is processed just once. For this purpose, we use the memoization – just like with recursive algorithms. Also, the resulting state of the memo table serves as the output of the algorithm. After the algorithm finishes, it contains pairs of the form $(A, \mathrm{NTP}(A))$.

5.1.3 Applications in Reliability Analysis

The two presented algorithms serve as an essential tool for the probabilistic evaluation described in section 1.5.1. For example, let us consider the calculation of system availability (1.17) of a system described by a structure function represented by diagram D using the *bottom-up* approach:

$$A^{\geq j} = \textsc{CalculateNTPPostStep}(\textsc{Root}(D), \{a \mid j \leq a < m\}), \tag{5.6}$$

and the *top-down* approach.

$$\begin{gathered} memo = \textsc{CalculateNTPLevel}(D) \\ A^{\geq j} = \sum_{a=j}^{m-1} \textsc{Lookup}(memo, T_a). \end{gathered} \tag{5.7}$$

As another example, let us also consider the calculation of system state probability (1.19) using the *bottom-up* approach:

$$\Pr\{\phi(\boldsymbol{x}) = j\} = \textsc{CalculateNTPPostStep}(\textsc{Root}(D), \{j\}), \tag{5.8}$$

and the *top-down* approach:

$$\begin{gathered} memo = \textsc{CalculateNTPLevel}(D) \\ \Pr\{\phi(\boldsymbol{x}) = j\} = \textsc{Lookup}\big(memo, T_j\big). \end{gathered} \tag{5.9}$$

Both approaches allow us to calculate the sum of NTPs of terminal nodes as well as individual NTPs. The difference between the approaches is that the *bottom-up* approach outputs the sum directly whilst the *top-down* approach outputs individual NTPs (stored in the $memo$) and the sum needs to be calculated additionally. This means that when we want to know individual NTPs, we need to run the *bottom-up* algorithm multiple times. The question is what the difference between the performance of the two approaches in different use cases is.

5.1.4 Experimental Comparison of the Approaches

The description of the two approaches suggests that the *bottom-up* approach could be faster (since it is simpler) in situations where we are interested in the sum of NTPs of terminal nodes e.g., in the calculation of system availability whereas the *top-down* approach could be faster in situations where we need to calculate NTPs of individual terminal nodes e.g., in the calculation of system state probabilities.

To verify the assumptions, we performed an experimental comparison of the two approaches presented in the paper [101]. In the experiment, we generated random diagrams using approaches described in section 4.1.1 and section 4.1.2 – properties of the generated diagrams can be found in Tab. 5.1. For each combination of the parameters, we generated 1,000 random diagrams. The subsequent comparison aimed to compare the approaches in the following use cases:

- calculation of all system state probabilities (Tab. 5.2);
- calculation of system availability with respect to the state $j = 1$ (Tab. 5.3).

In addition, we also aimed to evaluate different implementations of the priority queue used in the top-down algorithm. Specifically, we considered the following implementations:

- heap – de facto standard implementation in standard libraries of programming languages (top-down heap column) [102];
- bucket queue using an array list [103] of array lists to implement the buckets (top-down array column);
- bucket queue using an array list of linked lists to implement the buckets (top-down linked column).

The results for the calculation of system state probabilities are presented in Tab. 5.2. The relative performance of the two algorithms differs for different values of m. For $m = 3$, we can see that the *bottom-up* algorithm performs better even though we needed to run it three times – one time for each $j = 0,1,2$. On the other hand, for $m = 5$ we can see that the *top-down* algorithm performs better. The assumption is that it would also perform better for higher values of m as well, since we only need to run it once regardless of the number of system states. Another notable observation is that the bucket queue implementations outperform general implementation (heap) and that the array list implementation of the buckets is faster in larger diagrams.

In the second comparison – the calculation of $A^{\geq 1}$ – we considered only the cases where $m = 5$ since the calculation of system availability involves only a single invocation of the algorithm with both approaches regardless of the number of states. Hence, we expected that the simpler *bottom-up* algorithm would perform better – as the results of the first comparison suggested. Results of the comparison are presented in Tab. 5.3. They confirmed our assumption that a single invocation of the *bottom-up* algorithm is notably faster than a single invocation *top-down* algorithm. Finally, the results also reinforce the observation that the implementation of the bucket queue that uses an array list for the representation of the buckets has the best performance.

Tab. 5.1 Properties of diagrams generated for the experiment

Generating algorithm	n	m	**Average number of nodes**
Series-Parallel	50,000	3	280,365
Series-Parallel	10,000	5	347,204
Min-Max	40	3	13,528,106
Min-Max	20	5	2,220,734

Tab. 5.2 The average time in milliseconds required to calculate all system state probabilities

n	m	**Bottom-up**	**Top-down heap [ms]**	**Top-down array [ms]**	**Top-down linked [ms]**
50,000	3	15	22	20	18
10,000	5	58	47	30	32
40	3	1,621	3,975	1,925	2,840
20	5	573	647	330	430

Tab. 5.3 The average time in milliseconds required to calculate system availability $A^{\geq 1}$

n	m	**Bottom-up**	**Top-down heap [ms]**	**Top-down array [ms]**	**Top-down linked [ms]**
10,000	5	13	39	26	28
20	5	167	731	383	498

The results showed that both algorithms find applications in probabilistic analysis. As we assumed, the *top-down* algorithm is preferable in cases where we need to quantify NTPs in individual nodes. On the other hand, the simpler *bottom-up* algorithm is more advantageous in cases where we are only interested in the sum of NTPs. The results also showed that the *top-down* algorithm is a suitable use case for bucket-queue, which significantly outperforms the general implementation.

5.2 Probabilistic Calculations with Time-dependent Probabilities

Until now, in section 3.3.3 and section 5.1, we have only focused on the time-independent branch of probabilistic analysis. However, component state probabilities usually evolve in time and, therefore, it is necessary to consider this behavior in the probabilistic analysis to be able to describe and analyze systems more precisely.

In this section, we describe probabilistic analysis techniques that account for the component state probabilities no longer being constant numbers but rather expressions depending on variable t representing time. The expressions typically represent cumulative distribution functions of some probability distribution – such as exponential or Weibull [1] – that describe the probability that the component has failed (in the case of BDD) or that the component is in a state less than j (in the case of MSS) at time t. Consequently, the input of the probabilistic calculation is a matrix $\mathbb{P}_{n,m}$ of such expressions. We have identified two principal approaches to the calculation – the *basic* and the *symbolic* approaches. In the rest of this section, we proceed with the description and comparison of the two approaches.

5.2.1 Basic Approach

The *basic* approach is the simpler one of the two approaches. The first step carried out before the evaluation of the diagram is to first evaluate each element of $\mathbb{P}_{n,m}$ in time t transforming it into $\mathbb{P}_t$ – a simple matrix of floating-point numbers representing component state probabilities at time t. Then we proceed with the probabilistic calculations using the standard time-independent algorithms – either the *bottom-up* or *top-down* described in section 5.1. The basic approach requires no modification of the two algorithms and therefore can be used with existing tools. For example, the authors in [104] utilize this approach in the analysis of distributed generation power systems. However, a possible disadvantage of this approach is that it requires repeated evaluation of the diagram for each time point t. Alg. 5.1 illustrates usage of the *basic* approach in the evaluation of system availability at multiple time points.

```
function EVALUATEBASIC(diagram, j, timePoints, ℙ)
    for ∀ $t \in$ timePoints do
        $\mathbb{P}_t$ ← EVALUATEDISTRIBUTIONS(ℙ, t)
        values ← $\{a \mid j \leq a < m\}$
        $A^{\geq j}(t)$ ← CALCULATENTPPOSTSTEP(diagram, values, $\mathbb{P}_t$)
    end for
end function
```

Alg. 5.1 Basic approach to the calculation of system availability in multiple time points

5.2.2 Symbolic Approach

5.2.2.1 Description of the Symbolic Approach

The second approach utilizes symbolic expressions – hence the name symbolic approach. Various computer algebra systems such as Matlab, GNU Octave, or wxMaxima allow manipulation, evaluation, and analysis of expressions represented by trees. Fig. 5.1 shows a simple example of such a tree. Thus, the main idea of the symbolic approach is to perform the calculation on expressions rather than probabilities evaluated in time t. Therefore, the input matrix $\mathbb{P}_{n,m}$ contains symbolic expressions representing component state probabilities dependent on a single variable t representing time.

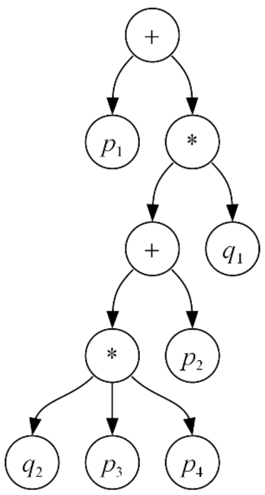


Fig. 5.1 Expression tree representing an expression that describes the availability of a BSS

Implementation of the symbolic approach requires a suitable representation of the expression trees. In our library, we chose GiNaC [105] – an open-source C++ library for (besides other use cases) the creation, manipulation, and evaluation of symbolic expressions. The input of our implementation is a matrix $\mathbb{P}_{n,m}$ of GiNaC expressions. Since GiNaC overloads standard arithmetic operators the manipulation of the expressions is very convenient. We can even reuse the code for the algorithms Alg. A.7 or Alg. A.8 by using techniques of generic programming – specifically the template mechanism of the C++ language.

The key difference from the basic approach is that after the last step of NTP calculation (Alg. A.7 or Alg. A.8), the result is a function in the form of an expression describing the probability. This expression contains a single variable – symbol t representing time. Now, to evaluate the probability at time t, we evaluate the expression for a given value of t. Thus, with the basic approach we evaluate BDD using the NTP calculation algorithm for each time point whereas with the symbolic approach, we run the NTP calculation algorithm only once, and then we evaluate the expression for each time point. Alg. 5.2 illustrates the usage of the *symbolic* approach in the evaluation of system availability at multiple time points.

```
function EVALUATESYMBOLIC(diagram, j, timePoints, $\mathbb{P}$)
    exprTree ←CREATETREE(diagram, $\mathbb{P}$)
    for ∀ t ∈ timePoints do
        $A^{\geq j}(t)$ ←EVALUATETREE(exprTree, t)
    end for
end function
```

Alg. 5.2 Symbolic approach to the calculation of system availability in multiple time points

An interesting question is which approach is better if we need to evaluate the probability at multiple time points. We provide an experimental comparison of the two approaches that investigates the relative performance difference in section 5.2.3.

5.2.2.2 Symbolic Computation Example

Let us consider a simple storage system analyzed in [106]. The system consists of two units connected in parallel. Each unit has two hard drives configured as RAID1 and RAID0 respectively. We will consider the system as a BSS for simplicity, and we will calculate its reliability (1.27). The topology of the system can be seen in Fig. 5.2. The system has the following structure function:

$$\phi(\boldsymbol{x}) = (x_1 \vee x_2) \vee x_3 x_4. \tag{5.10}$$

Using the structure function, component reliabilities (1.24), and the inclusion-exclusion principle we can calculate the system reliability using the following formula:

$$\begin{aligned} R(t) = p_1(t) + p_2(t) + p_3(t)p_4(t) \\ - p_3(t)p_4(t)\big(p_1(t) + p_2(t) + p_3(t)p_4(t)\big). \end{aligned} \tag{5.11}$$

By using the bottom-up algorithm, we obtain the following formula:

$$R(t) = q_1(t)\big(p_2(t) + q_2(t)p_3(t)p_4(t)\big) + p_1(t), \tag{5.12}$$

which, after substituting $1 - p_i(t)$ for each $q_i(t)$, agrees with the formula (5.11). Let us assume the same exponential distributions of component reliabilities as authors in [106] – we can see the distributions in Tab. 5.4. If we substitute the distributions into expression (5.12) we can plot the system reliability function which we can see in Fig. 5.3.

Tab. 5.4 Storage system component reliabilities

Component	**Component reliability $p_i(t)$**
1	$\exp(-t/25{,}359)$
2	$\exp(-t/6{,}246)$
3	$\exp(-t/4{,}764)$
4	$\exp(-t/44{,}360)$

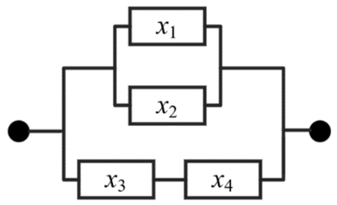


Fig. 5.2 Reliability block diagram depicting topology of a simple storage system

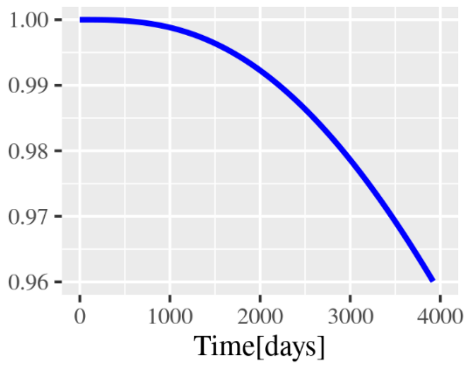


Fig. 5.3 Reliability function of the storage system with the topology depicted in Fig. 5.2

Besides the difference in the approaches described in section 5.2.2.1, the symbolic approach offers more flexibility such as that it allows for easier interaction with computer algebra systems – we can serialize the expression and import it into some computer algebra system for further analysis. For example, we obtained the expression (5.12) by running the bottom-up algorithm with a matrix containing symbols $q_i(t)$ and $p_i(t)$. Furthermore, we obtained the chart in Fig. 5.3 by exporting the expression importing it into the R [107] system, and using the ggplot [108] library to create the chart.

5.2.3 Comparison of Symbolic and Basic Approaches

We performed an experimental comparison of the basic approach and symbolic approach to determine which approach performs better in the evaluation of time-dependent system reliability at multiple time points. We performed three experiments using our TeDDy library which implements both approaches.

5.2.3.1 Storage System Example

The first comparison we performed was on the storage system presented in section 5.2.3.1. In the experiment, we evaluated system reliability using component reliabilities presented in Tab. 5.4 at 10; 100; 1,000; and 10,000 selected time points. Tab. 5.5 shows the result of the comparison. The durations in the table were obtained as average from 100 replications of the computation. Column *Basic computation* contains the total time in nanoseconds required to compute system reliability at the given number of time points. Column *Symbolic init* contains the time needed to create the expression tree and column *Symbolic computation* the total time in nanoseconds required to compute system reliability at the given number of time points. The results clearly show that the basic approach is in the orders of magnitude faster than the symbolic approach even when we need to evaluate a higher number of time points.

Tab. 5.5 Comparison of the basic and symbolic approach in the computation of system reliability of a four-component storage system

Time points	Basic computation [ns]	Symbolic init [ns]	Symbolic computation
10	956	9,252	267,071
100	7,258	9,454	2,607,722
1,000	69,447	9,949	25,945,398
10,000	692,683	13,456	257,718,581

5.2.3.2 Random Series-parallel Systems

The second comparison aims to compare the two approaches in the analysis of series-parallel systems with different topologies. For this purpose, we generated random series-parallel systems with 10, 20, 30, and 40 components using the approach described in section 4.1.2. For each such system, we computed system reliability in 10 time points. Since the systems are randomly generated, we assume exponential distributions of component reliabilities with randomly generated rate parameters. Table. 3 contains the results of the comparison. The durations in the table were obtained for each variable count n as average from 10 randomly generated system topologies and 10 replications for each topology. In addition to the previously described columns the table also contains |*BDD*| and |*Tree*| columns which contain the average number of nodes in BDD and the expression trees respectively.

The results confirm the results of the first experiment that the basic approach is significantly faster. Moreover, the results also indicate that the complexity of the expression tree increases dramatically with increasing number of variables. This suggests that the symbolic approach is not suitable for a system with a higher number of components while the basic approach seems to scale very well if the size of the BDD stays reasonable.

Tab. 5.6 Comparison of the basic and symbolic approach in the computation of system reliability of randomly generated series-parallel systems

n	|BDD|	|Tree|	Basic computation [ns]	Symbolic init [ns]	Symbolic computation [ns]
10	12	599	1,739	26,187	3,823,367
20	22	15,218	3,606	51,791	101,280,004
30	32	546,208	6,020	82,222	3,595,178,608
40	42	11,494,828	7,401	103,151	72,100,562,769

5.2.3.3 PLA Benchmark Circuits

The two experiments that we described so far used series-parallel systems. Therefore, in the last experiment, we decided to analyze systems of different nature – PLA circuits from the IWLS’93 benchmark [93]. Reliability analysis of logic circuits is specific since the structure function contains variables representing inputs of the circuits as well as variables representing unreliable logic gates [14]. The analysis aims only at the variables representing the logic gates fixing the input variables for all possible inputs. Hence, the size of the BDD is relatively small despite a higher number of variables. In this experiment, we also assumed exponential distributions of component reliabilities. Just like in the first experiment, we evaluated system reliability at 10; 100; 1,000; and 10,000 time points.

Tab. 5.7 presents the results of the experiment. Additionally the PLA file column contain the name of the benchmark, and the n column contains the number of variables representing the logic gates – the number of variables in the analyzed BDD. The results show that, again, the *basic* approach performed better than the *symbolic* approach. However, the relative difference between the two approaches is much smaller.

Tab. 5.7 Comparison of the basic and symbolic approach in the computation of system reliability of PLA circuits

PLA file	***n***	**Time points**	**Basic computation [ns]**	**Symbolic init [ns]**	**Symbolic computation [ns]**
con1	11	10	1,905	986	36,352
con1	11	100	18,002	1,078	357,491
con1	11	1,000	178,788	1,275	3,566,846
con1	11	10,000	1,791,139	2,017	35,643,194
xor5	17	10	2,210	698	23,954
xor5	17	100	21,317	763	237,646
xor5	17	1,000	212,002	874	2,384,453
xor5	17	10,000	2,120,591	1,391	23,797,647
rd53	35	10	3,860	2,158	74,064
rd53	35	100	37,498	2,338	731,114
rd53	35	1,000	374,347	2,709	7,277,917
rd53	35	10,000	3,736,264	4,274	72,870,141
squar5	40	10	4,469	8,019	220,688
squar5	40	100	43,178	8,306	2,189,516
squar5	40	1,000	430,617	9,086	21,934,328
squar5	40	10,000	4,297,874	14,722	218,934,519
sqrt8	44	10	4,864	2,000	67,313
sqrt8	44	100	47,538	2,220	661,964
sqrt8	44	1,000	473,333	2,555	6,612,181
sqrt8	44	10,000	4,760,873	4,303	66,284,840

Each of the above-described experiments showed that the basic approach performs much better than the symbolic approach if we consider the speed of the evaluation of NTP. Although the results are specific to our implementation – our library TeDDy and GiNaC library for the manipulation of expressions – the relative difference between the two approaches is considerable and therefore is unlikely to change significantly for other implementations. However, the symbolic approach that we presented is still a valid and useful tool for time-dependent reliability analysis because of the mentioned possibilities to further manipulate and analyze the expression.

Conclusion

This thesis dealt with the application of decision diagrams in the reliability analysis of complex systems. It provided a comprehensive overview of the steps of the reliability analysis process with a focus on the algorithms operating on decision diagrams representing structure functions. Several new algorithms were introduced, and various improvements and generalizations of existing algorithms were provided. The contributions of the thesis are the results of solving the following research problems:

- analysis of existing approaches and algorithms utilized in the representation of the structure function by decision diagram and their subsequent analysis:
  - ✓ Chapter 1 introduced general approaches used in reliability analysis,
  - ✓ Chapter 2 described discrete functions as the mathematical foundation of the structure function (section 2.1) and means of their analysis (section 2.2) and representation (section 2.4);
  - ✓ finally, Chapter 3 dealt with decision diagrams and their applications in reliability analysis (section 3.3).
- implementation of a performant and robust software library for the creation and manipulation of decision diagrams:
  - ✓ Chapter 3 presented essential aspects of a software library for the creation and manipulation of decision diagrams (section 3.2);
  - ✓ all the described algorithms and techniques are implemented in our open-source library TeDDy [75].
- evaluation, adjustment, and improvement of existing algorithms for the creation and manipulation of decision diagrams;
  - ✓ Chapter 4 provided an experimental comparison of different ways of the order of evaluation of the *apply* algorithm (section 4.2);
  - ✓ Chapter 4 evaluated the per-state representation of series-parallel systems (section 4.3);
  - ✓ Chapter 3 introduced an algorithm for the calculation of system state frequencies (section 3.3.2) and Chapter 4 showed that it is preferable also in the special case of BSS (section 4.4);
- creation of new decision diagram algorithms and methods specialized for the use case of topological and probabilistic reliability analysis:

- ✓ Chapter 3 presented a generalized version of the *apply* algorithm (section 3.2.3) for the dynamic creation of decision diagrams and Chapter 4 showed that the algorithm is suitable for practical use cases (section 4.2.2)
- ✓ Chapter 4 introduced a new universal algorithm for the efficient calculation of arbitrary logic derivatives (section 4.5).

In conclusion, the major contribution of this thesis is the description of the optimization of the creation and manipulation of decision diagrams for the reliability analysis of large complex systems. The description involves existing techniques and algorithms as well as new algorithms proposed in this thesis. Finally, a notable practical contribution is the open-source software library specialized in reliability analysis with decision diagrams.

Resume

1 Predmet výskumu

Analýza spoľahlivosti je dôležitou súčasťou životného cyklu takmer všetkých systémov. Je dôležitá už vo fáze návrhu systémov, kedy nám pomáha zostrojiť systém tak, aby dokázal plniť požadovanú funkcionalitu dostatočne dlhý čas s požadovanou spoľahlivosťou. Nemenej dôležitá je aj pri plánovaní údržby systémov alebo pri identifikácii komponentov kritických pre fungovanie systému.

Prvým krokom analýzy je identifikácia počtu stavov systému. Ďalším krokom je vytvorenie matematického popisu systému. V tejto práci sa zameriavame na popis systému tzv. štruktúrnou funkciou [2]. Štruktúrna funkcia priradí danému stavu komponentov stav systému – popisuje závislosť stavu systému na stave jeho komponentov. Vo všeobecnosti je štruktúrna funkcia diskrétnou funkciou. Jej konkrétna forma závisí od počtu stavov systému a od počtu stavov komponentov systému.

Skúmaním vlastností štruktúrnej funkcie získavame informácie o vlastnostiach skúmaného systému. Jednou z vlastností, ktoré štruktúrna funkcia preberá je komplexnosť systému. Tá môže byť spôsobená napríklad veľkých počtom komponentov systému, rôznou povahou komponentov alebo komplikovanými vzťahmi medzi komponentami. Obzvlášť pri veľkom počte komponentov je preto potrebné štruktúrnu funkciu efektívne reprezentovať. Vhodná reprezentácia musí zvládnuť popísať aj rozsiahle systémy a musí tiež umožňovať efektívne spracovanie v počítači.

Rozhodovací diagram [11], [12] je štruktúra, ktorá spĺňa obe uvedené vlastnosti. Ide o acyklický graf, ktorý bol navrhnutý na efektívnu reprezentáciu diskrétnych funkcií. Rozhodovacie diagramy sa vo všeobecnosti považujú za veľmi efektívny spôsob reprezentácie štruktúrnej funkcie. Povaha komplexných systémov a neustály nárast zložitosti však vytvárajú tlak na neustále zlepšovanie existujúcich techník a navrhovanie nových prístupov. Voľne dostupné softvérové nástroje v súčasnosti poskytujú algoritmy na všeobecnú prácu s rozhodovacími diagramami. Algoritmy na analýzu spoľahlivosti sú však často implementované iba pre konkrétny prípad použitia alebo nie sú voľne dostupné.

Hlavným cieľom tejto práce je preto *optimalizácia aplikácie rozhodovacích diagramov pri analýze spoľahlivosti zložitých systémov,* z čoho vyplývajú nasledujúce výskumné témy:

- analýza existujúcich prístupov a algoritmov využívaných pri reprezentácii štruktúrnej funkcie pomocou rozhodovacieho diagramu a pri následnej analýze;
- implementácia výkonnej a robustnej softvérovej knižnice na tvorbu a manipuláciu s rozhodovacími diagramami zameranej na využitie diagramov v analýze spoľahlivosti;
- návrh, úprava a zlepšenie existujúcich algoritmov na tvorbu a manipuláciu s rozhodovacími diagramami;
- vytvorenie nových algoritmov a metód založených na využití rozhodovacích diagramov špecializovaných na analýzu spoľahlivosti.

2 Analýza spoľahlivosti

2.1 Počet stavov systému

Pred začiatkom analýzy systému je potrebné identifikovať počet stavov systému. Najjednoduchší prístup je popisovať iba dva stavy systému – systém *funguje* a systém *zlyhal* – ktoré popisujeme číslami 1 a 0 v tomto poradí. Takto popísaný systém nazývame dvojstavový systém (BSS z angl. „Binary-State System“) [1], [2]. Pre systémy, ktoré sú zo svojej povahy dvojstavové je takýto prístup postačujúci. Príkladom takéhoto systému je logický obvod. Rovnako je vhodný aj pre systémy, v ktorých môže aj veľmi malé zhoršenie stavu spôsobiť škody na technike alebo ohroziť zdravie ľudí. V takomto prípade môže ísť napríklad o riadiaci systém elektrárne.

Mnohé systémy však dokážu plniť svoju úlohu aj po zhoršení ich stavu. Príkladom môže byť transportná sieť, ktorá funguje s menšou prenosovou kapacitou. Takéto systémy nazývame viacstavové (MSS z angl. „Multi-State System“) [3]. Stavy takýchto systémov popisujeme číslami 0 pre stav, v ktorom systém nefunguje až po číslo $m - 1$ pre stav, v ktorom systém funguje bez obmedzení, kde m je celkový počet stavov. Pri tomto type systémov ďalej rozlišujeme *homogénne* systémy, v ktorých je počet stavov všetkých komponentov a počet stavov systému rovnaký a *nehomogénne* systémy, v ktorých môžu mať rôzne komponenty a celý systém rozdielny počet stavov. Táto vlastnosť je typická pre systémy zložené z komponentov rôznej povahy – napr. z technických zariadení a ľudí.

Výhodou BSS je ich jednoduchosť a stým spojená jednoduchosť modelov, ktoré ich popisujú – či už z pohľadu veľkosti modelov alebo z pohľadu výpočtovej zložitosti algoritmov. Výhodou je tiež dostupnosť väčšieho množstva algoritmov a nástrojov. Na druhej strane, ich nevýhodou môže byť až prílišné zjednodušenie v prípade popisu systémov, ktoré nie sú vo svojej povahe dvojstavové. V takomto prípade je pre získanie presnejších výsledkov potrebné popisovať takéto systémy ako viacstavové – avšak za cenu zložitejšieho modelu a väčšej výpočtovej zložitosti.

2.2 Štruktúrna funkcia

Štruktúrna funkcia je zobrazenie, ktoré každému stavu komponentov priradí prislúchajúci stav systému. Vo všeobecnosti ide o diskrétnu funkciu, ktorá má v prípade *nehomogénneho* MSS nasledovnú podobu [5]:

$$\begin{aligned}\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x})\colon \{0,1,\dots,m_1-1\} \times \dots \times \{0,1,\dots,m_n-1\} \\ \rightarrow \{0,1,\dots,m-1\},\end{aligned} \qquad \{1\}$$

kde n je počet komponentov systému, x_i popisuje stav i-teho komponentu pre $i = 1,2,\dots,n$; m je počet stavov systému, m_i je počet stavov i-teho komponentu a $\boldsymbol{x} = (x_1, x_2, \dots, x_n)$ je stavový vektor, ktorý obsahuje stav všetkých komponentov.

Definícia $\{1\}$ popisuje najvšeobecnejší prípad a súhlasí s definíciou celočíselnej funkcie [10]. V špeciálnom prípade *homogénneho* MSS, kedy platí $m_i = m_j = m$ pre $i, j = 1,2,\dots,n$, má nasledovnú, jednoduchšiu, podobu [5]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x})\colon \{0,1,\dots,m-1\}^n \rightarrow \{0,1,\dots,m-1\}, \qquad \{2\}$$

ktorá je zhodná s definíciou viachodnotovej logickej funkcie [10]. Ak navyše platí, že $m = 2$, štruktúrna funkcia popisuje BSS a má nasledovnú formu [2]:

$$\phi(x_1, x_2, \dots, x_n) = \phi(\boldsymbol{x})\colon \{0,1\}^n \rightarrow \{0,1\}, \qquad \{3\}$$

ktorá je zhodná s definíciou booleovskej funkcie [8].

Z definícií $\{1\}$, $\{2\}$ a $\{3\}$ je zrejmé, že definícia $\{3\}$ predstavuje najvšeobecnejší prípad a definície $\{1\}$ a $\{2\}$ predstavujú iba špeciálne prípady. V ďalšom popise budeme preto uvažovať štruktúrnu funkciu vo forme $\{3\}$.

2.3 Logické derivácie

Skúmaním vlastností štruktúrnej funkcie získavame informácie o systéme, ktorý popisuje. Logický diferenciálny počet [10], [109] – podobne ako klasický diferenciálny počet – umožňuje skúmať dynamické vlastnosti diskrétnych funkcií. Dôležitým nástrojom pre analýzu spoľahlivosti sú tzv. logické derivácie, ktoré popisujú, ako sa mení hodnota funkcie pri konkrétnej zmene hodnoty premennej.

Základnou deriváciou je smerová logická derivácia celočíselnej funkcie $f(\boldsymbol{x})$ podľa premennej x_i, ktorú definujeme nasledovným spôsobom [10]:

$$\frac{\partial f(j \to h)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{ak } f(s_j, \boldsymbol{x}) = j \text{ a } f(r_j, \boldsymbol{x}) = h \\ 0, & \text{inak,} \end{cases} \qquad \{4\}$$

kde $s, r, j, h \in \{0,1, \dots, m-1\}, s \neq r$ a $j \neq h$. Notácia $f(s_j, \boldsymbol{x})$ predstavuje kofaktor funkcie f podľa premennej x_j s hodnotou s. Kofaktor je funkcia $n-1$ premenných, ktorú získame zafixovaním hodnoty premennej x_j na hodnotu s. Podobne je derivácia $\{4\}$ funkcia $n-1$ premenných, ktorá nadobúda hodnotu 1 iba v bodoch, v ktorých zmena hodnoty premennej x_i z hodnoty s na hodnotu r spôsobí zmenu hodnoty funkcie f z hodnoty j na hodnotu h.

Derivácia $\{4\}$ popisuje jednu špecifickú zmenu hodnoty funkcie. Pri celočíselnej funkcii však existuje vzhľadom na prípustné hodnoty s, r, j, h relatívne veľký počet konkrétnych derivácií, ktoré je možné vyhodnotiť. Keďže jednotlivé derivácie popisujú iba zlomok všetkých situácií, ich použitie by bolo pomerne nepraktické. Pre získanie obsiahlejšieho pohľadu na správanie funkcie preto používame integrované smerové logické derivácie [30]. Literatúra popisuje tri typy týchto derivácií a to:

- **typ I** definovaný nasledovne:

$$\frac{\partial f(j \searrow)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{ak } f(s_i, \boldsymbol{x}) = j \text{ a } f(s_i, \boldsymbol{x}) < j \\ 0, & \text{inak,} \end{cases} \qquad \{5\}$$

- **typ II** definovaný nasledovne:

$$\frac{\partial f(\searrow)}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{ak } f(s_i, \boldsymbol{x}) > f(r_i, \boldsymbol{x}) \\ 0, & \text{inak,} \end{cases} \qquad \{6\}$$

- a **typ III** definovaný nasledovne:

$$\frac{\partial f(h_{\geq j} \to h_{<j})}{\partial x_i(s \to r)} = \begin{cases} 1, & \text{ak } f(s_i, \boldsymbol{x}) \geq j \text{ a } f(r_i, \boldsymbol{x}) < j \\ 0, & \text{inak.} \end{cases} \qquad \{7\}$$

Definície $\{5\}$, $\{6\}$ a $\{7\}$ sa v rámci jednotlivých typov môžu líšiť napr. smerom zmeny hodnoty funkcie. Povaha zmeny sa však pre konkrétny typ nemení.

Zmena hodnoty premennej a následná zmena hodnoty štruktúrnej funkcie zodpovedajú zmene stavu komponentu a následnej zmene stavu systému. Logické derivácia preto predstavujú veľmi silný nástroj pre skúmanie vplyvu komponentov na stav systému. V nasledujúcich sekciách preto popíšeme využitie derivácií pri výpočte rôznych ukazovateľov spoľahlivosti.

2.4 Topologická analýza

Štruktúrna funkcia popisuje topológiu systému, na základe ktorej dokážeme vykonať topologickú analýzu. Základným topologickým ukazovateľom je relatívna frekvencia stavov systému vzhľadom na stav j definovaná nasledovne [110]:

$$Fr^{\geq j} = \mathrm{TD}(\phi(\boldsymbol{x}) \geq j), \tag{8}$$

kde $\phi(\boldsymbol{x})$ je štruktúrna funkcia, $j \in \{0,1,\dots,m-1\}$ a notácia $\mathrm{TD}(.)$ označuje tzv. hustotu pravdivosti argumentu – relatívny počet vstupných vektorov, pre ktoré argument (funkcia s booleovským výstupom) nadobúda hodnotu 1. Frekvencia stavov systému tak popisuje relatívny počet možných stavov komponentov, pre ktoré je systém v stave j alebo v stave lepšom ako j. Frekvenciu stavov systému môžeme použiť na jednoduché porovnanie dvoch rôznych konfigurácií systému napríklad vo fáze návrhu systému.

Frekvencia stavov systému popisuje celý systém jedným číslom a nehovorí nič o vplyve jednotlivých komponentov systému. Takúto informáciu poskytuje jeden z tzv. ukazovateľov dôležitosti [7] zvaný štruktúrna dôležitosť. Štruktúrnu dôležitosť (SI z angl. „Structural Importance“) je možné definovať viacerými spôsobmi. Z pohľadu vyhodnocovania je veľmi výhodná definícia pomocou logickej derivácie [20]:

$$\mathrm{SI}_i = \mathrm{TD}\left(\frac{\partial f\left(h_{\geq j} \to h_{<j}\right)}{\partial x_i(s \to r)}\right). \tag{9}$$

SI_i popisuje relatívny počet situácií kedy zmena stavu i-teho komponentu zo stavu s do stavu r spôsobí zmenu stavu systému popísanú deriváciou. V definícii {9} sme použili integrovanú smerovú logickú deriváciu typu III {7}. Pre výpočet SI je však možné použiť aj ostatné typy derivácií. Presný význam SI potom závisí od použitej derivácie.

2.5 Pravdepodobnostná analýza

Nevýhodou topologickej analýzy je, že predpokladá rovnakú pravdepodobnosť stavov komponentov. Stavy komponentov sa však v praxi vyskytujú s rôznymi pravdepodobnosťami. Pre získanie presnejších charakteristík systému je preto potrebné vziať do úvahy aj pravdepodobnosti stavov komponentov. Ďalšou dôležitou vlastnosťou je, že pravdepodobnosť stavov komponentov sa v čase mení. Preto rozlišuje časovo závislé a časovo nezávislé pravdepodobnostné charakteristiky.

Časovo nezávislé pravdepodobnosti stavov komponentov označujeme nasledovne:

$$p_{i,k} = \Pr\{x_i = k\}, \tag{10}$$

kde $i = 1,2,\dots,n$ a $k = 0,1,\dots,m_i - 1$. Notácia {10} označuje pravdepodobnosť, že komponent i je v stave k. Časovo závislé pravdepodobnosti stavov komponentov označujeme podobne:

$$p_{i,s}(t) = \Pr\{Z_i(t) = s\}, \tag*{\{11\}}$$

kde $s = 0,1,\dots m_i - 1$, premenná t reprezentuje čas a funkcia $Z_i(t)$ popisuje stav i-teho komponentu v čase t.

Základnou pravdepodobnostnou charakteristikou systému je *dostupnosť* vzhľadom na stav systému j. Časovo nezávislú dostupnosť MSS definujeme nasledovne [3]:

$$A^{\geq j}(\boldsymbol{p}) = \Pr\{\phi(\boldsymbol{x}) \geq j\}, \tag*{\{12\}}$$

kde $j \in \{1,2,\dots,m-1\}$ a $\boldsymbol{p}$ je matica pravdepodobností stavov komponentov. Dostupnosť {12} zodpovedá pravdepodobnosti, že systém je v stave j alebo v lepšom stave. Komplementárnym ukazovateľom k dostupnosti je nedostupnosť systému vzhľadom na stav j definovaná nasledovne [3]:

$$U^{\geq j}(\boldsymbol{p}) = \Pr\{\phi(\boldsymbol{x}) < j\}, \tag*{\{13\}}$$

ktorá zodpovedá pravdepodobnosti, že systém je v stave horšom ako j.

Podobne ako frekvencia stavov systému {8} popisujú dostupnosť a nedostupnosť celého systému jednou pravdepodobnosťou. Keďže však môžu byť rôzne komponenty rôzne spoľahlivé, neponúkajú žiadnu informáciou o dôležitosti jednotlivých komponentov. Takúto informáciou poskytujú rôzne ukazovatele dôležitosti. Jedným z bežne používaných ukazovateľov je Birnbaumova dôležitosť (BI z angl. „Birnbaum importance“). Podobne ako pri SI {9} je z praktického hľadiska výhodná definícia pomocou logickej derivácie [20]:

$$\mathrm{BI}_{i,s}^{\geq} = \Pr\left\{\frac{\partial\phi\left(h_{\geq j} \to h_{<j}\right)}{\partial x_i(s \to s-1)} \leftrightarrow 1\right\}. \tag*{\{14\}}$$

$\mathrm{BI}_{i,s}^{\geq}$ udáva pravdepodobnosť, že zhoršenie stavu i-teho komponentu zo stavu s to stavu $s - 1$ spôsobí zmenu stavu systému popisovanú deriváciou. Notácia $. \leftrightarrow 1$ zodpovedá jav kedy ľavý argument nadobúda hodnotu 1. V definícii {14} sme použili integrovanú smerovú logickú deriváciu typu III {7}. Pre výpočet BI je však možné použiť aj ostatné typy derivácií. Presný význam BI potom závisí od použitej derivácie.

Definície časovo závislých verzií vyššie popísaných pravdepodobnostných ukazovateľov sú podobné ich časovo nezávislým ekvivalentom. Zásadným rozdielom je však argument, ktorý už nie je jednoduchá matica pravdepodobností, ale je ním vektor stavových funkcií.

Časovo závislá dostupnosť systému vzhľadom na stav systému j je definovaná nasledovne [3]:

$$A^{\geq j}(t) = \Pr\{\phi(\boldsymbol{Z}(t)) \geq j\}, \tag{15}$$

kde $\boldsymbol{Z}(t) = (Z_1(t), Z_2(t), \dots, Z_n(t))$ je vektor stavových funkcií jednotlivých komponentov.

3 Rozhodovacie diagramy

Pre zostrojenie a vyhodnotenie štruktúrnej funkcie je potrebné vybrať vhodný spôsob jej reprezentácie. Reprezentácia musí umožniť efektívne spracovanie v počítači a zároveň umožniť efektívne reprezentovať aj rozsiahlejšie funkcie popisujúce komplexné systémy. Rozhodovací diagram je grafová štruktúra navrhnutá na reprezentáciu diskrétnych funkcií, ktorá spĺňa obe uvedené vlastnosti. Binárny rozhodovací diagram [11] (BDD z angl. „Binary Decision Diagram“) je najjednoduchší typ rozhodovacieho diagramu navrhnutý na reprezentáciu booleovských funkcií. Jeho zovšeobecnením je viachodnotový rozhodovací diagram [12] (MDD z angl. „Multi-valued Decision Diagram“) navrhnutý na reprezentáciu viachodnotových logických funkcií {2} a celočíselných funkcií {1}. Vo zvyšku textu budeme uvažovať najvšeobecnejší prípad MDD, ktorý reprezentuje celočíselnú funkciu.

3.1 Štruktúra diagramu

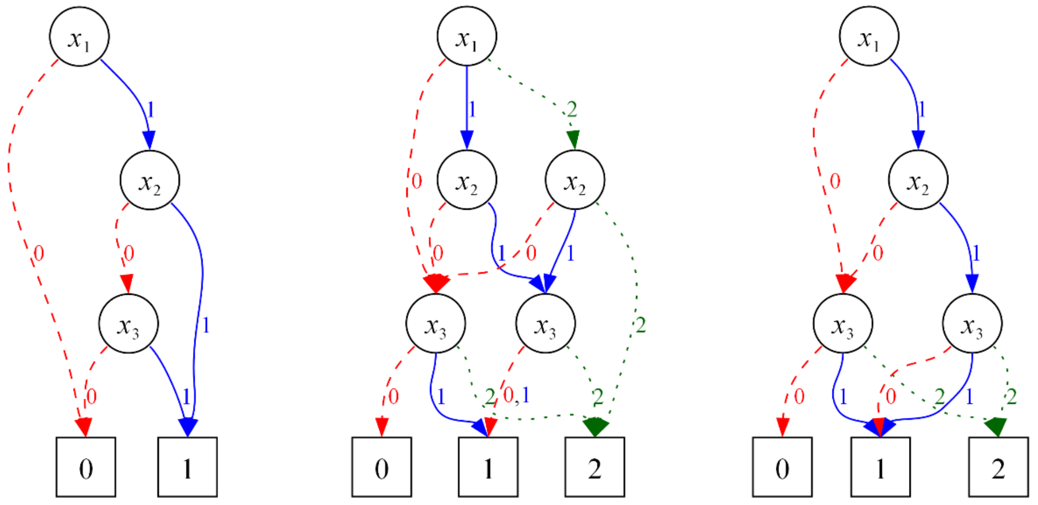
MDD je orientovaný acyklický graf, ktorý sa skladá z *vnútorných* vrcholov, ktoré reprezentujú premenné a *koncových* vrcholov, ktoré reprezentujú hodnoty funkcie. Vrcholy diagramu sú uložené na úrovniach. Jedna úroveň obsahuje vnútorné vrcholy reprezentujúce rovnakú premennú, s výnimkou poslednej úrovne, ktorá obsahuje koncové vrcholy. Vnútorný vrchol reprezentujúci premennú x_i má m_i výstupných hrán, ktoré vedú do vrcholov na nižších úrovniach. Koncový vrchol nemá žiadne výstupné hrany a počet koncových vrcholov je najviac m. Ukážku všetkých štruktúry rôznych typov MDD môžeme vidieť na Obr. 1.

3.2 Tvorba diagramov

Kľúčovou vlastnosťou MDD je unikátnosť a zdieľanie vrcholov. Tieto vlastnosti sú prakticky zabezpečené udržiavaním tzv. *tabuľky unikátnych vrcholov* – pred vytvorením nového vrcholu sa najprv kontroluje, či už požadovaný vrchol neexistuje v tabuľke.

Základom vytvorenia MDD je tzv. *priama* tvorba – MDD reprezentujúci funkciu jednej premennej alebo konštantnú funkciu môžeme vytvoriť jednoducho bez potreby sofistikovanejšieho algoritmu. MDD reprezentujúci komplikovanejšie funkcie môžeme následne získať spájaním priamo vytvorených diagramov pomocou binárnych operácií ($\wedge, \vee, \oplus_m$, min, max, ...). Takýto prístup nazývame *dynamická* tvorba MDD. Na spájanie používame algoritmus *apply* [11], ktorého vstupom sú dva MDD a binárna operácia a výstupom je nový MDD, ktorý reprezentuje novú funkciu získanú spojením vstupných funkcií binárnou operáciou. Opakovaným použitím algoritmu *apply* tak môžeme vytvoriť MDD reprezentujúci ľubovoľnú funkciu.

Pre úplnosť spomenieme ďalší možný prístup k tvorbe MDD zvaný *statická* tvorba, ktorá spočíva v transformácii pravdivostnej tabuľky na MDD [80]. Tento prístup je vhodný pre tvorbu menších MDD, ktoré môžu slúžiť ako vstup do procesu dynamickej tvorby.



Obr. 1 Vľavo: BDD reprezentujúci booleovskú funkciu, uprostred: MDD reprezentujúci viachodnotovú logickú funkciu, vpravo: MDD reprezentujúci celočíselnú funkciu

3.3 Zefektívnenie tvorby diagramov

Vytvorenie MDD reprezentujúceho štruktúrnu funkciu je nutný krok k následnej analýze systému. V prípade komplexných systémov môže byť štruktúrna funkcia rozsiahla a komplikovaná, preto je potrebné vytvorenie diagramu vykonať čo najefektívnejšie. Časť práce sa preto venuje zefektívneniu a zjednodušeniu tohto procesu.

3.3.1 Zovšeobecnenie spájania diagramov

Algoritmus *apply* môžeme považovať za binárnu operáciu. Mnohé funkcie/podsystémy, ktoré chceme popisovať sú však d-árne (kde $d \in \mathbb{N}$). Príkladom môže byť trojvstupové logické hradlo (hradlo AND na Obr. 2) alebo paralelný (pod)systém (Obr. 2). Bežne používaným riešením je viacnásobné použitie binárneho algoritmu *apply*. V prípade logického obvodu na Obr. 2 by tak volanie *apply* mohlo vyzerať nasledovne:

$$\textsc{Apply}(\textsc{Apply}(\textsc{Apply}(X_1, X_2, \wedge), X_3, \wedge), X_4, \vee),$$

kde zápis X_i predstavuje MDD reprezentujúci funkciu jednej premennej x_i. Oveľa prirodzenejšie by však bolo volanie *apply* nasledovným spôsobom:

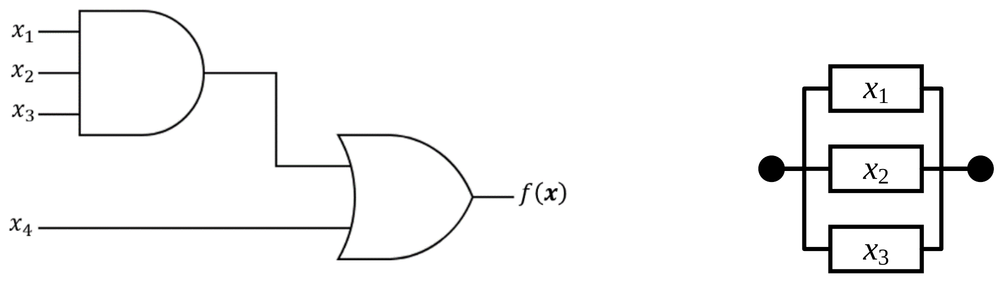
$$\textsc{Apply}(\mathbf{Apply}(\boldsymbol{X_1}, \boldsymbol{X_2}, \boldsymbol{X_3}, \wedge), X_4, \vee).$$

Vo zvýraznenej časti výrazu môžeme vidieť pomyselnú ternárnu verziu *apply*.

V práci sme preto navrhli zovšeobecnenú verziu algoritmu *apply*, ktorú sme pomenovali *extended apply*. Podobne ako základná verzia algoritmu je tento postavený na vzťahu, ktorý popisuje vnútorný vrchol diagramu pomocou Shannonovej expanzie [49]. Nami zovšeobecnený vzťah má nasledovnú podobu:

$$\odot_d\,(f_1, f_2, \dots, f_d)(\boldsymbol{x}) = \sum_{k=0}^{m_i-1} \Big(\{x_i \leftrightarrow k\} * \big(\odot_d\,(f_1, f_2, \dots, f_d)(k_i, \boldsymbol{x})\big)\Big), \qquad \{16\}$$

kde výraz $\{x_i \leftrightarrow k\}$ predstavuje tzv. logický bikondicionál, ktorý nadobúda hodnotu 1 práve vtedy a len vtedy, ak premenná x_i nadobúda hodnotu k a $\odot_d$ je d-árna asociatívna operácia. Členy súčtu sa dajú stotožniť s výstupnými hranami vnútorného vrcholu a celý výraz s vnútorným vrcholom MDD.



Obr. 2 Príklady systémov, na popis ktorých je vhodnejšie použiť ternárne funkcie; vľavo jednoduchý logický obvod; vpravo paralelný systém s tromi komponentami

Navrhnutý algoritmus sme experimentálne porovnali so základnou verziou [95]. V experimente sme vytvárali MDD reprezentujúce náhodné d-cestné stromy pomocou nášho algoritmu *extended apply* s rôznou aritou. Výsledky porovnania môžem vidieť v Tab. 1.

Výsledky porovnania ukazujú, že základná verzia dosahuje lepšie výsledky v porovnaní s rozšírenými verziami – a teda použitím rozšíreného algoritmu nedosahujeme výrazné zrýchlenie. Na druhej strane, verzia s aritou 3 sa ukázala byť rovnako rýchla a efektívnejšia z hľadiska počtu krokov algoritmu, čo naznačuje, že môžu existovať prípady použitia, v ktorých je rozšírená verzia vhodnejšia.

Tab. 1 Priemerný čas v milisekundách potrebný na vytvorenie MDD z výrazového stromu popisujúceho sériovo-paralelný systém zložený z n_{nax} komponentov

n_{max}	d			
	2	**3**	**4**	**5**
20 000	79	78	94	113
40 000	177	176	209	252
60 000	280	278	333	401
80 000	384	383	457	550
100 000	500	495	592	712

3.3.2 Poradie spájania diagramov

Pri tvorbe diagramov je často potrebné spojiť rádovo desiatky až stovky MDD rovnakou operáciou $\odot$. Jedno použitie algoritmu *extenden apply* pri takto vysokom počte MDD nie je vhodné kvôli veľkej výpočtovej náročnosti analyzovanej v texte práce. Riešením je preto viacnásobné použitie či už základnej alebo rozšírenej verzie *apply*. Kvôli asociativite operácie $\odot$ je možné toto spojenie vykonať mnohými spôsobmi. V práci sme analyzovali prístup zvaný *left-fold*[5], v ktorom diagramy spájame sekvenčne zľava doprava:

$$\Big(\big((D_1 \odot D_2) \odot D_3\big) \odot D_4\Big) \odot D_5$$

a prístup *tree-fold*, v ktorom diagramy spájame hierarchicky postupne po dvojiciach:

$$\big((D_1 \odot D_2) \odot (D_3 \odot D_4)\big) \odot D_5,$$

kde notácia D_i predstavuje počiatočný MDD.

V práci sme skúmali, či a ako poradie spájania ovplyvňuje rýchlosť vytvorenia diagramu [111], [112]. V Tab. 2 môžeme vidieť výsledky jedného z experimentálnych porovnaní, v ktorom sme vytvárali BDD reprezentujúce binárne sčítačky popísané PLA súbormi [91].

Výsledky experimentu ukázali, že v konkrétnom prípade použitého benchmakru dokáže poradie spájania výrazne ovplyvniť rýchlosť vytvorenia výsledného diagramu.

[5] Názvy *left-fold* a *tree-fold* pochádzajú z programovacích jazykov, kde sa funkcie s danými menami používajú na spracovanie údajových štruktúr [88]

V tomto konkrétnom prípade sa prístup *tree-fold* (ktorý je menej intuitívny) ukázal ako výrazne rýchlejší. Je však dôležité spomenúť, že v ďalší experimentoch s inými dátami sa, naopak, prístup *left-fold* ukázal ako rýchlejší. Z výsledkov našich experimentov preto usudzujeme, že pre nástroje na prácu s diagramami je výhodné implementovať oba prístupy.

Tab. 2 Priemerný čas v milisekundách potrebný na vytvorenie BDD reprezentujúcich výstupy binárnej sčítačky

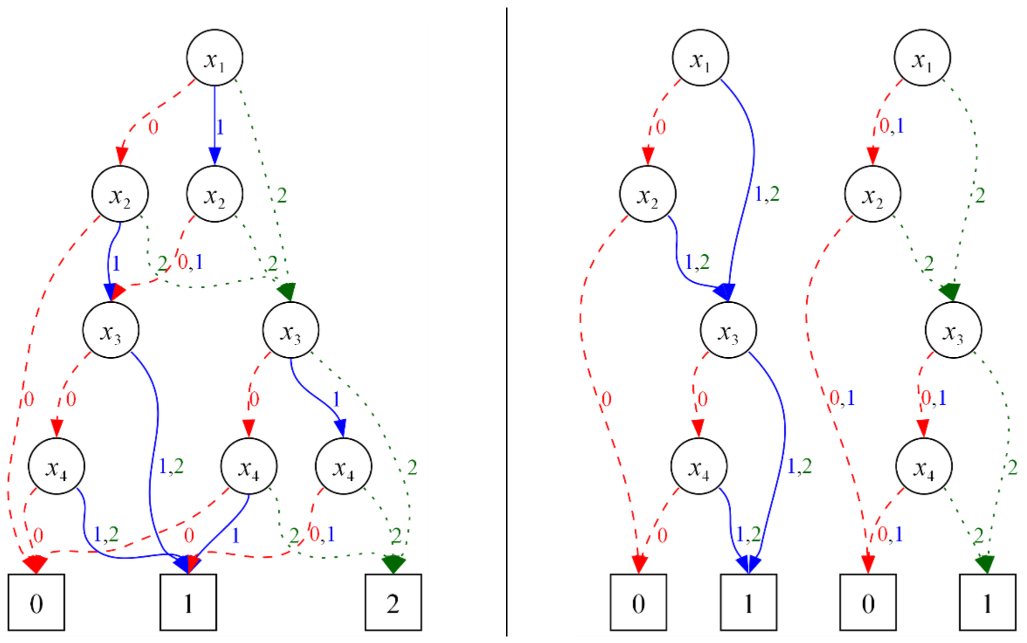
Počet bitov sčítačky	**Počet spájaných diagramov**	**Left fold [ms]**	**Tree fold [ms]**
10-adder-col	10 191	163	71
11-adder-col	20 427	477	180
12-adder-col	40 911	1 828	448
13-adder-col	81 867	5 342	1 084
14-adder-col	163 783	16 579	2 753

4 Aplikácia rozhodovacích diagramov v analýze spoľahlivosti

Výsledky, ktoré sme popísali v predchádzajúcej sekcii je možné aplikovať na tvorbu diagramov vo všeobecnosti. Práca je však zameraná na špecifické využitie diagramov v analýze spoľahlivosti. Vo zvyšku textu sa preto venujeme tejto problematike.

4.1 Reprezentácia štruktúrnej funkcie

Primárne a pre ďalšiu analýzu nevyhnutné využitie diagramov v analýze spoľahlivosti spočíva v reprezentácii štruktúrnej funkcie. Základný a intuitívny prístup spočíva v reprezentácii celej štruktúrnej funkcie $\phi(\boldsymbol{x})$ jedným MDD. V prípade MSS však existuje aj alternatívny prístup, kedy každý stav systému popíšeme individuálne funkciami $\phi(\boldsymbol{x}) \geq 1, \phi(\boldsymbol{x}) \geq 2, \dots \phi(\boldsymbol{x}) \geq m-1$ [86]. Vizuálne porovnanie týchto dvoch prístupov môžeme vidieť na Obr. 3. Nevýhodou tohto prístupu je potreba vytvorenia niekoľkých MDD namiesto jedného. Na druhej strane, výhodou môže byť zjednodušenie popisu jednotlivých stavov systému, keďže každý je popísaný samostatne.



Obr. 3 Štruktúrna funkcia $\phi(\boldsymbol{x})$ reprezentovaná jedným diagramom (vľavo) a sériou diagramov (vpravo) pozostávajúcej z funkcií $\phi(\boldsymbol{x}) \geq 1$ a $\phi(\boldsymbol{x}) \geq 2$ v tomto poradí (vpravo)

Zaujímavou otázkou je tiež porovnanie veľkosti resp. celkového počtu vrcholov MDD potrebných na reprezentáciu systému pri použití daných prístupov. Na zodpovedanie tejto otázky sme vykonali experiment, v ktorom sme porovnávali oba prístupy pri reprezentácii náhodne generovaných sériovo-paralelných systémov – ktoré považujeme za komplexné pri veľkom počte komponentov, kedy je veľkosť reprezentácie obzvlášť dôležitá. Výsledky porovnania môžeme vidieť v Tab. 3.

Tab. 3 Priemerný počet vrcholov v jednom MDD a v sérii MDD v závislosti od počtu komponentov systému (n) v prípade homogénnych sériovo-paralelných 3, 4 a 5 stavových MSS

n	**Jeden MDD**			**Séria MDD**		
	3	4	5	3	4	5
500	1 995	5 138	10 756	1 002	1 502	2 002
1 000	4 232	11 436	24 926	2 002	3 002	4 002
1 500	6 562	18 210	40 591	3 002	4 502	6 002
2 000	8 959	25 346	57 424	4 002	6 002	8 002
2 500	11 406	32 749	75 128	5 002	7 502	10 002

Výsledky experimentu jednoznačne ukázali, že v prípade sériovo-paralelných systémov je reprezentácia pomocou série diagramov výrazne výhodnejšia ako použitie jedného diagramu. Tento výsledok je konzistentný aj pre systémy rôznych veľkostí s rozdielnym počtom stavov.

4.2 Výpočet frekvencie stavov systému

Frekvencia stavov systému {8} je jedna zo základných topologických charakteristík systému, ktorou dokážeme jednoducho porovnať systémy s rôznymi topológiami. Jej výpočet spočíva vo vyhodnotení jednoduchého zlomku:

$$Fr^{\geq j} = \frac{\alpha_{\phi,j}}{\alpha_{\phi}}, \qquad \{17\}$$

kde $\alpha_{\phi,j}$ je počet stavových vektorov (situácií), kde je systém v stave $\geq j$ a α_{ϕ} je celkový počet stavových vektorov. Na vyčíslenie čitateľa môžeme využiť exitujúci algoritmus *satisfy-count* [11] a menovateľ vypočítame jednoduchým súčinom $\prod_{i=1}^{n} m_i$. Praktický problém takéhoto výpočtu je, že aj keď je výsledok z intervalu $[0,1]$, čitateľ a menovateľ zlomku sú často čísla, ktoré nie je možné reprezentovať 64-bitovými údajovými typmi.

V prípade BSS je možné problém s limitovaným rozsahom údajových typov vyriešiť vypočítaním logaritmov čitateľa a menovateľa a jednoduchou úpravou vzťahu {17}, ktorý popisujeme v práci. Tento postup sa však nedá aplikovať vo všeobecnosti na MSS. V práci sme preto popísali špecializovanú verziu existujúceho algoritmu na výpočet pravdepodobností, ktorá nie je limitovaná rozsahom údajových typov. Otázkou zostávalo, či má zmysel aplikovať tento algoritmus aj na BSS alebo či je v tomto prípade postup využívajúci logaritmy rýchlejší. Na zodpovedanie otázky sme vykonali experimentálne porovnanie spomínaných prístupov vrátane základného na BDD reprezentujúcich náhodne generované systémy. Výsledky porovnania môžeme vidieť v Tab. 4.

Tab. 4 Priemerný čas v mikrosekundách potrebný na výpočet frekvencie stavov systému pomocou rôznych prístupov

n	**Satisfy-count [µs]**	**Satisfy-count-log [µs]**	**Náš algoritmus [µs]**
10	12	14	10
30	1 387	1 823	1 337
60	59 157	69 047	57 107
80	[6] 471 942	191 991	161 950
90	[6] 788 309	329 563	287 166
100	[6] 1 057 991	470 303	407 762

[6] S použitím knižnice GMP [87] pre výpočty s neobmedzenou presnosťou

Výsledky experimentu ukázali, že náš algoritmus funguje lepšie ako prístup využívajúci logaritmy. Potvrdil aj náš predpoklad, že hoci je možné použiť základný prístup založený na algoritme *satisfy-count* – ktorý má porovnateľný výkon pre $n < 63$ – výpočty zahŕňajúce celé čísla s neobmedzenou presnosťou sú podstatne pomalšie. Preto sme dospeli k záveru, že je lepšie použiť náš algoritmus aj pre špeciálny prípad BSS.

4.3 Efektívny výpočet logických derivácií

Logické derivácie sú veľmi užitočný nástroj pre výpočet mnohých ukazovateľov spoľahlivosti, ako napr. rôznych ukazovateľov dôležitosti [20] alebo napríklad minimálnych rezných vektorov [23]. Ich efektívny výpočet je preto pre proces analýzy komplexných systémov veľmi dôležitý.

Deriváciu funkcie reprezentovanej MDD môžeme relatívne jednoducho vypočítať nasledovaním jej definície (napr. {7}) a s použitím existujúcich algoritmov na manipuláciu MDD konkrétne *cofactor* [11], *transform* a *apply* [11]. Tento prístup funguje pomerne dobre, avšak využitie všeobecných algoritmov nemôže naplno využiť špecifiká výpočtu derivácií. Jeho nevýhodou je tiež že, v základnej podobe nie je univerzálny – pre výpočet rôznych derivácií musíme uvedené algoritmy kombinovať iným spôsobom s rôznymi parametrami.

V práci sme preto navrhli špecializovaný algoritmus na výpočet ľubovoľnej derivácie [98]. Náš algoritmus vychádza z nasledujúcej univerzálnej definície logickej derivácie:

$$\frac{\partial f(\Lambda)}{\partial x_i(s \to r)} = \Lambda\big(f(s_i, \boldsymbol{x}), f(r_i, \boldsymbol{x})\big), \qquad \{18\}$$

kde notácia Λ predstavuje funkciu, ktorá popisuje, či zmena funkcie daná jej argumentami predstavuje požadovanú zmenu – v tom prípade vráti hodnotu 1 – alebo nie a vráti hodnotu 0. Funkcia Λ je parametrom nášho algoritmu, ktorý tak môžeme použiť na výpočet ľubovoľnej logickej derivácie.

Po technickej stránke náš algoritmus kombinuje algoritmy *cofactor, transform* a *apply*, ktoré sa v základom prístupe používajú samostatne, do jedného kroku. Na základe tejto vlastnosti sme predpokladali, že náš algoritmus by mal byť vo výpočet derivácií rýchlejší. Na kvantifikovanie rozdielu v rýchlosti sme preto vykonali experimentálne porovnanie základného prístupu a nášho algoritmu pri výpočte integrovanej derivácie typu III {7} z MDD reprezentujúcich náhodne generované systémy. Výsledky porovnania môžeme vidieť v Tab. 5.

Tab. 5 Priemerný čas v milisekundách potrebný na výpočet IDPLD typu III pre každú premennú pomocou základného postupu a pomocou nášho algoritmu

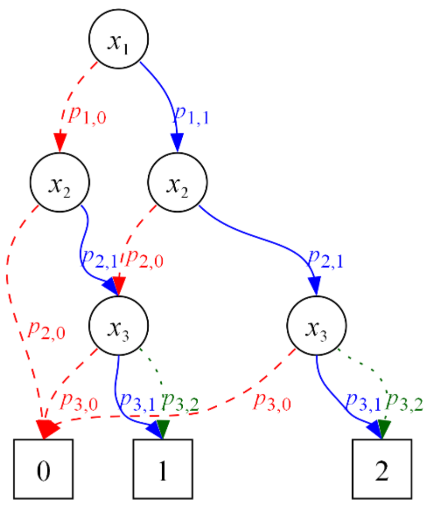
m	n	**Počet vrcholov**	**Základný prístup [ms]**	**Náš algoritmus [ms]**	**Náš algoritmus / základný prístup**
2	32	128 322	3 570	1 538	0,43079
3	23	531 698	6 005	2 978	0,49597
4	20	1 591 344	18 540	9 625	0,51917
5	17	1 401 163	13 208	6 874	0,52042

Experimentálne porovnanie ukazuje, že náš algoritmus je približne o 50 % rýchlejší ako základný prístup. Keďže výpočet logických derivácií je jedným zo základných krokov analýzy spoľahlivosti a dôležitosti komponentov, náš algoritmus môže výrazne urýchliť proces analýzy komplexných systémov.

4.4 Pravdepodobnostné vyhodnotenie diagramov

Pravdepodobnostná analýza poskytuje v porovnaní s jednoduchou topologickou analýzou oveľa presnejší popis správania systému a vplyvu jednotlivých komponentov prostredníctvom ukazovateľov akými sú napr. dostupnosť systému {12} alebo Birnbaumova dôležitosť {14} a mnohých iných. Pri použití MDD na reprezentáciu štruktúrnej funkcie je pri vyhodnocovaní všetkých pravdepodobnostných ukazovateľov kľúčovou úlohou výpočet pravdepodobnosti navštívenia [48] koncového vrcholu MDD (NTP z angl. „Node Traversing Probability“). MDD je našťastie pre výpočet pravdepodobností veľmi výhodnou štruktúrou. Pravdepodobnosti stavov komponentov môžeme pomyselne stotožniť s hranami MDD, ako môžeme vidieť na Obr. 4. Výpočet pravdepodobností je následne záležitosťou vhodnej prehliadky diagramu a násobenia vhodných pravdepodobností.

V literatúre existujú dva zásadné prístupy/algoritmy k výpočtu NTP zvané *bottom-up* a *top-town*. Tieto dva prístupy sa líšia typom prehliadky, ktorú pri výpočte používajú. Preto ich v práci označujeme aj ako *post-order* a *level-order* algoritmy. Jedným z výsledkov prezentovaných v práci je porovnanie týchto prístupov pri výpočte rôznych pravdepodobnostných ukazovateľov. Výsledky nášho porovnania môžeme vidieť v Tab. 6.



Obr. 4 Pravdepodobnostný rozhodovací diagram s pravdepodobnosťami stavov komponentov znázornenými na hranách diagramu

Algoritmy sme porovnávali pri výpočte pravdepodobností všetkých stavov m-stavového systému zloženého z n-komponentov so sériovo-paralelnou topológiou (prvé dva riadky tabuľky) a s náhodnou topológiou (posledné dva riadky tabuľky). Výsledky ukázali, že pri vyššom počte stavov systému je výhodnejšie použiť *top-down* algoritmus a, naopak, pri nižšom počte stavov systému je výhodnejší *bottom-up* algoritmus. Ďalším rozdielom, ktorý v práci popisujeme, a ktorý je potrebné pri výbere algoritmu zvážiť je, že jedno vykonanie *top-down* algoritmu umožňuje vypočítať individuálne pravdepodobnosti stavov systému ako aj dostupnosť pre rôzne úrovne. Na druhej strane jednoduchší *bottom-up* algoritmus umožňuje pri jednom vykonaní výpočet iba jednej charakteristiky.

Tab. 6 Priemerný čas v milisekundách potrebný na výpočet pravdepodobnosti všetkých stavov systému

n	m	**Bottom-up**	**Top-down [ms]**
50 000	3	15	20
10 000	5	58	30
40	3	1 621	1 925
20	5	573	330

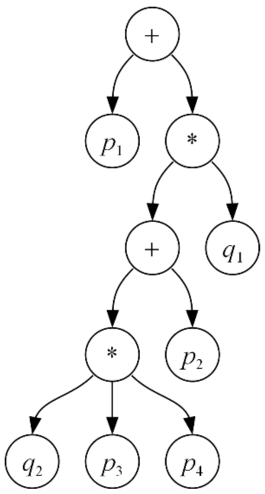
4.5 Časovo závislé pravdepodobnostné výpočty

Vo vyššie popísanom porovnaní prístupov k výpočtu NTP sme pracovali s konštantnými pravdepodobnosťami stavovo komponentov. Posledná časť práce sa venuje modifikácií uvedených algoritmov, ktorá umožní pracovať aj s pravdepodobnosťami stavov komponentov, ktoré už nie sú jednoduchými konštantami, ale sú funkciami času.

Prvým riešeným problémom je výpočet NTP (ktorá súhlasí s vybraným pravdepodobnostným ukazovateľom) v mnohých časových. Na riešenie tohto problému sme identifikovali dva prístupy, ktoré sme nazvali *základný* a *symbolický*. *Základný* prístup využíva algoritmus *bottom-up* alebo *top-down* bez modifikácií. V rámci dodatočného kroku však potrebuje vyhodnotiť všetky pravdepodobnosti stavov komponentov v danom čase t. Tento prístup môžeme stručne zosumarizovať nasledovným pseudokódom, ktorý prezentuje funkciu na výpočet dostupnosti systému vo všetkých časových okamihoch uložených v zozname *timePoints*:

```
function EVALUATEBASIC(diagram, j, timePoints, $\mathbb{P}$)
    for $\forall$ $t \in$ timePoints do
        $\mathbb{P}_t \leftarrow$ EVALUATEDISTRIBUTIONS($\mathbb{P}$, $t$)
        values $\leftarrow$ $\{a \mid j \le a < m\}$
        $A^{\ge j}(t) \leftarrow$ CALCULATENTPPOSTSTEP(diagram, values, $\mathbb{P}_t$)
    end for
end function.
```

Symbolický prístup je založený na využití symbolických výpočtov, podľa ktorých nesie svoje pomenovanie. Podobne ako *základný* prístup využíva jeden z dvojice algoritmov. Tento však musí byť upravený alebo vhodne implementovaný tak, aby dokázal sčítavať a násobiť symbolické výrazy, ktoré môže byť reprezentované napríklad výrazovým stromom. Príklad takéhoto stromu môžeme vidieť na Obr. 5.



Obr. 5 Výrazový strom reprezentujúci výraz, ktorý popisuje dostupnosť BSS; premenné p_i a q_i reprezentujú pravdepodobnosti, že i-ty komponent funguje a nefunguje v tomto poradí

Symbolický prístup tak najprv využije jeden z algoritmov na získanie výrazového stromu reprezentujúceho vybraný pravdepodobnostný ukazovateľ (napr. dostupnosť systému). Vstupom algoritmu je v tomto prípade matica symbolických výrazov. Následne už vyhodnocuje iba získaný strom v každom časovom okamihu. Podobne ako pri základom prístupe môžeme symbolický prístup zosumarizovať nasledovným pseudokódom:

```
function EVALUATESYMBOLIC(diagram, j, timePoints, ℙ)
    exprTree ←CREATETREE(diagram, ℙ)
    for ∀ t ∈ timePoints do
        $A^{\geq j}(t)$ ←EVALUATETREE(exprTree, t)
    end for
end function.
```

V našej knižnici TeDDy [75] sme symbolický prístup implementovali pomocou knižnice GiNaC [105], ktorá podporuje prácu so symbolickými výrazmi v jazyku C++.

Zaujímavou otázkou je ako sa dva uvedené prístupy líšia z pohľadu časovej náročnosti na výpočet pravdepodobnostného ukazovateľa v mnohých časových okamihoch. Za účelom preskúmania tohto rozdielu sme vykonali experimentálne porovnanie týchto prístupov pri výpočte dostupnosti náhodne generovaných sériovo-paralelných BSS. Výsledky tohto porovnania môžeme vidieť v Tab. 7. Stĺpce |BDD| a |Strom| obsahujú veľkosť danej štruktúry. Zvyšné stĺpce obsahujú celkový priemerný čas v nanosekundách potrebný na vyhodnotenie dostupnosti systému v 10 časových okamihoch.

Tab. 7 Porovnanie *základného* a *symbolického* prístupu pri výpočte dostupnosti náhodne generovaných sériovo-paralelných systémov

n	|BDD|	|Strom|	Základný prístup [ns]	Vytvorenie stromu [ns]	Symbolické výpočty [ns]
10	12	599	1 739	26 187	3 823 367
20	22	15 218	3 606	51 791	101 280 004
30	32	546 208	6 020	82 222	3 595 178 608
40	42	11 494 828	7 401	103 151	72 100 562 769

Výsledky experiment ukázali, že základný prístup funguje oveľa lepšie ako symbolický prístup, ak berieme do úvahy rýchlosť vyhodnotenia NTP. Podobné výsledky sme získali aj z iných experimentov, ktoré vyhodnocovali tisícky rôznych časových okamihov. Hoci výsledky sú špecifické pre našu implementáciu – knižnicu TeDDy a knižnicu GiNaC na manipuláciu s výrazmi –relatívny rozdiel medzi oboma prístupmi je značný. Preto je nepravdepodobné, že by sa pri iných implementáciách výrazne zmenil.

S*ymbolický* prístup dosahuje v porovnaní so *základným* pomerne zlé výsledky. Na druhej strane nám však poskytuje možnosti, ktoré nemôžeme *základným* prístupom dosiahnuť. Jednou z nich je napríklad možnosť získať výraz popisujúci pravdepodobnostný ukazovateľ. Výraz môžeme analyzovať napríklad použitím knižnice GiNaC alebo ho môžeme exportovať do systému ako napr. Matlab alebo wxMaxima.

5 Záver

Práca sa zaoberala aplikáciou rozhodovacích diagramov pri analýze spoľahlivosti komplexných systémov. Poskytla obsiahly prehľad krokov procesu analýzy spoľahlivosti so zameraním na algoritmy založené na využití rozhodovacích diagramov reprezentujúcich štruktúrnu funkciu. Ďalej predstavila niekoľko nových algoritmov a poskytla rôzne vylepšenia a zovšeobecnenia existujúcich algoritmov. Prínosom práce sú výsledky riešenia nasledujúcich výskumných problémov:

- Analýza existujúcich prístupov a algoritmov používaných pri reprezentácii štruktúrnej funkcie pomocou rozhodovacích diagramov a ich následná analýza:
    - ✓ v úvodnej časti práca predstavila všeobecné prístupy používané pri analýze spoľahlivosti;
    - ✓ ďalej opísala diskrétne funkcie ako matematický základ štruktúrnej funkcie a spôsoby ich analýzy a reprezentácie;
    - ✓ nakoniec úvodnej časti sa zaoberala rozhodovacími diagramami a ich aplikáciami v analýze spoľahlivosti.

- Implementácia výkonnej a robustnej softvérovej knižnice na tvorbu a manipuláciu s rozhodovacími diagramami:
    - ✓ prvá časť jadra práce predstavila základné aspekty softvérovej knižnice na tvorbu a manipuláciu s rozhodovacími diagramami;
    - ✓ všetky algoritmy a techniky opísané v práci boli implementované v open-source knižnici TeDDy.
- Vyhodnocovanie, úprava a zlepšovanie existujúcich algoritmov na tvorbu a manipuláciu s rozhodovacími diagramami;
    - ✓ praktická časť práce poskytla experimentálne porovnanie rôznych spôsobov poradia vyhodnocovania algoritmu aplikácie;
    - ✓ ďalej vyhodnotila rôzne prístupy k reprezentácii štruktúrnej funkcie sériovo-paralelných systémov;
    - ✓ práca tiež predstavila univerzálny algoritmus na výpočet stavových frekvencie stavov systému.

- Vytvorenie nových algoritmov a metód rozhodovacích diagramov špecializovaných na prípad použitia topologickej a pravdepodobnostnej analýzy spoľahlivosti:
    - ✓ práca predstavila zovšeobecnenú verziu algoritmu na dynamickú tvorbu rozhodovacích diagramov;
    - ✓ nakoniec predstavila nový univerzálny algoritmus na efektívny výpočet ľubovoľných logických derivácií.

Záverom možno konštatovať, že hlavným prínosom tejto práce je opis optimalizácie tvorby a manipulácie s rozhodovacími diagramami pre analýzu spoľahlivosti rozsiahlych komplexných systémov. Opis zahŕňa existujúce techniky a algoritmy, ako aj nové algoritmy navrhnuté v tejto práci. Napokon, významným praktickým prínosom je softvérová knižnica s otvoreným zdrojovým kódom špecializovaná na analýzu spoľahlivosti pomocou rozhodovacích diagramov.

Bibliography

[1] M. Rausand and A. Høyland, *System Reliability Theory, 2nd ed.* Hoboken, NJ: John Wiley & Sons, Inc., 2004.

[2] E. Zio, *An Introduction to the Basics of Reliability and Risk Analysis*. World Scientific Publishing Company, 2007. doi: 10.1142/6442.

[3] B. Natvig, *Multistate Systems Reliability Theory with Applications*. Chichester, UK: John Wiley & Sons, Inc., 2011. doi: 10.1002/9780470977088.

[4] I. Frenkel, A. Lisnianski, and Y. Ding, *Multi-state System Analysis and Optimization for Engineers and Industrial Managers*. 2010. doi: 10.1007/978-1-84996-320-6.

[5] A. Lisnianski and G. Levitin, *Multi-State System Reliability: Assessment, Optimization and Application*. Singapore, SG: World Scientific Publishing Company, 2003. doi: 10.1142/5221.

[6] M. Kvassay and E. Zaitseva, *Topological analysis of multi-state systems based on direct partial logic derivatives*, vol. 0, no. 9783319634. 2018. doi: 10.1007/978-3-319-63423-4_14.

[7] W. Kuo and X. Zhu, "Importance Measures in Reliability, Risk, and Optimization: Principles and Applications," *Importance Measures in Reliability, Risk, and Optimization: Principles and Applications*, Apr. 2012, doi: 10.1002/9781118314593.

[8] Y. Crama and P. Hammer, *Boolean Functions: Theory, Algorithms, and Applications*. 2011. doi: 10.1017/CBO9780511852008.

[9] R. Stanković, J. Astola, and C. Moraga, *Representation of Multiple-Valued Logic Functions*. Morgan & Claypool, 2012. doi: 10.2200/S00420ED1V01Y201205DCS037.

[10] S. Yanushkevich, D. Miller, V. Shmerko, and R. Stankovic, *Decision Diagram Techniques for Micro- and Nanoelectronic Design Handbook*. Boca Raton, FL: CRC Press, 2006. doi: 10.1201/9781420037586.

[11] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986, doi: 10.1109/TC.1986.1676819.

[12] A. Srinivasan, T. Ham, S. Malik, and R. K. Brayton, "Algorithms for discrete function manipulation," in *1990 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, 1990, pp. 92–95. doi: 10.1109/ICCAD.1990.129849.

[13] M. R. Choudhury and K. Mohanram, "Reliability Analysis of Logic Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 3, pp. 392–405, 2009, doi: 10.1109/TCAD.2009.2012530.

[14] M. Kvassay, E. Zaitseva, V. Levashenko, and J. Kostolny, “Reliability Analysis of Multiple-Outputs Logic Circuits Based on Structure Function Approach,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 3, pp. 398–411, 2017, doi: 10.1109/TCAD.2016.2586444.

[15] Y. Watanabe, T. Oikawa, and K. Muramatsu, “Development of the DQFM method to consider the effect of correlation of component failures in seismic PSA of nuclear power plant,” *Reliab Eng Syst Saf*, vol. 79, no. 3, pp. 265–279, 2003, doi: https://doi.org/10.1016/S0951-8320(02)00053-4.

[16] B. Nystrom, L. Austrin, N. Ankarback, and E. Nilsson, “Fault Tree Analysis of an Aircraft Electric Power Supply System to Electrical Actuators,” in *2006 International Conference on Probabilistic Methods Applied to Power Systems*, 2006, pp. 1–7. doi: 10.1109/PMAPS.2006.360325.

[17] P. Praks, V. Kopustinskas, and M. Masera, “Probabilistic modelling of security of supply in gas networks and evaluation of new infrastructure,” *Reliab Eng Syst Saf*, vol. 144, pp. 254–264, 2015, doi: https://doi.org/10.1016/j.ress.2015.08.005.

[18] E. Zaitseva, V. Levashenko, M. Kvassay, and P. Barach, “Healthcare system reliability analysis addressing uncertain and ambiguous data,” in *2017 International Conference on Information and Digital Technologies (IDT)*, 2017, pp. 442–451. doi: 10.1109/DT.2017.8024334.

[19] E. Zaitseva, V. Levashenko, and M. Rusin, “Reliability analysis of healthcare system,” in *2011 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2011, pp. 169–175.

[20] E. Zaitseva and L. Vitaly, “Importance analysis by logical differential calculus,” *Automation and Remote Control*, vol. 74, 2013, doi: 10.1134/S000511791302001X.

[21] W. Kuo and X. Zhu, “Some recent advances on importance measures in reliability,” *IEEE Trans Reliab*, vol. 61, no. 2, pp. 344–360, 2012, doi: 10.1109/TR.2012.2194196.

[22] M. Kvassay, V. Levashenko, and E. Zaitseva, “Analysis of minimal cut and path sets based on direct partial Boolean derivatives,” *Proc Inst Mech Eng O J Risk Reliab*, vol. 230, no. 2, pp. 147–161, Apr. 2016, doi: 10.1177/1748006X15598722.

[23] M. Kvassay, E. Zaitseva, V. Levashenko, and J. Kostolny, “Minimal cut vectors and logical differential calculus,” *Proceedings of The International Symposium on Multiple-Valued Logic*, pp. 167–172, 2014, doi: 10.1109/ISMVL.2014.37.

[24] K. Kołowrocki, *Reliability of Large and Complex Systems*. 2014.

[25] J. E. Byun, H. M. Noh, and J. Song, “Reliability growth analysis of k-out-of-N systems using matrix-based system reliability method,” *Reliab Eng Syst Saf*, vol. 165, pp. 410–421, Sep. 2017, doi: 10.1016/J.RESS.2017.05.001.

[26] N. Mannai and S. Gasmi, “Optimal design of k-out-of-n system under first and last replacement in reliability theory,” *Operational Research*, vol. 20, no. 3, pp. 1353–1368, Sep. 2020, doi: 10.1007/S12351-018-0375-4/FIGURES/3.

[27] D. A. Marx and A. D. Slonim, “Assessing patient safety risk before the injury occurs: an introduction to sociotechnical probabilistic risk modelling in health care,” *Qual Saf Health Care*, vol. 12 Suppl 2, no. Suppl 2, Dec. 2003, doi: 10.1136/QHC.12.SUPPL_2.II33.

[28] W. S. Griffith, “Multistate reliability models,” *J Appl Probab*, vol. 17, no. 3, pp. 735–744, Sep. 1980, doi: 10.2307/3212967.

[29] G. Levitin, L. Podofillini, and E. Zio, “Generalised importance measures for multi-state elements based on performance level restrictions,” *Reliab Eng Syst Saf*, vol. 82, no. 3, pp. 287–298, Dec. 2003, doi: 10.1016/S0951-8320(03)00171-6.

[30] M. Kvassay, E. Zaitseva, and V. Levashenko, “Importance analysis of multi-state systems based on tools of logical differential calculus,” *Reliab Eng Syst Saf*, vol. 165, pp. 302–316, 2017, doi: https://doi.org/10.1016/j.ress.2017.03.021.

[31] T. Nakagawa, “Stochastic Processes,” 2011, doi: 10.1007/978-0-85729-274-2.

[32] D. Butler, “A complete importance ranking for components of binary coherent systems, with extensions to multi-state systems,” *Naval Research Logistics Quarterly*, vol. 26, no. 4, pp. 565–578, Dec. 1979, doi: 10.1002/NAV.3800260402.

[33] Z. W. Birnbaum, *On the Importance of Different Components in a Multicomponent System*. in Technical report (University of Washington. Laboratory of Statistical Research). Laboratory of Statistical Research, Department of Mathematics, University of Washington, 1968. [Online]. Available: https://books.google.sk/books?id=LRzcMgEACAAJ

[34] J. Barabady and U. Kumar, “Availability allocation through importance measures,” *undefined*, vol. 24, no. 6, pp. 643–657, 2007, doi: 10.1108/02656710710757826.

[35] J. D. Andrews and S. Beeson, “Birnbaum’s measure of component importance for noncoherent systems,” *IEEE Trans Reliab*, vol. 52, no. 2, pp. 213–219, Jun. 2003, doi: 10.1109/TR.2003.809656.

[36] F. C. Meng, “Comparing the importance of system components by some structural characteristics,” *IEEE Trans Reliab*, vol. 45, no. 1, pp. 59–65, 1996, doi: 10.1109/24.488917.

[37] E. Zio and L. Podofillini, “Monte Carlo simulation analysis of the effects of different system performance levels on the importance of multi-state components,” *Reliab Eng Syst Saf*, vol. 82, no. 1, pp. 63–73, Oct. 2003, doi: 10.1016/S0951-8320(03)00124-8.

[38] G. Levitin, L. Podofillini, and E. Zio, “Generalised importance measures for multi-state elements based on performance level restrictions,” *Reliab Eng Syst Saf*, vol. 82, no. 3, pp. 287–298, Dec. 2003, doi: 10.1016/S0951-8320(03)00171-6.

[39] J. E. Ramirez-Marquez and D. W. Coit, “Composite importance measures for multi-state systems with multi-state components,” *IEEE Trans Reliab*, vol. 54, no. 3, pp. 517–529, Sep. 2005, doi: 10.1109/TR.2005.853444.

[40] B. Steinbach and C. Posthoff, “Boolean Differential Calculus,” *Synthesis Lectures on Digital Circuits and Systems*, vol. 12, pp. 1–215, Aug. 2017, doi: 10.2200/S00766ED1V01Y201704DCS052.

[41] M. Kvassay, P. Rusnak, and P. Sedlacek, “Computation of Birnbaum’s Importance Using Logic Differential Calculus,” in *2019 42nd International Conference on Telecommunications and Signal Processing (TSP)*, 2019, pp. 613–616. doi: 10.1109/TSP.2019.8768854.

[42] M. Kvassay, E. N. Zaitseva, and V. G. Levashenko, “Minimal Cut and Minimal Path Vectors in Reliability Analysis of Binary- and Multi-State Systems,” in *ICTERI*, 2017.

[43] M. Kvassay, P. Rusnak, R. S. Stankovic, and A. Forgac, “Use of Binary Decision Diagrams in Importance Analysis Based on Minimal Cut Vectors,” *2019 14th International Conference on Advanced Technologies, Systems and Services in Telecommunications, TELSIKS 2019 - Proceedings*, pp. 78–81, Oct. 2019, doi: 10.1109/TELSIKS46999.2019.9002349.

[44] M. Kvassay, P. Rusnak, E. Zaitseva, and R. S. Stanković, “Multi-Valued Decision Diagrams in Importance Analysis Based on Minimal Cut Vectors,” in *2020 IEEE 50th International Symposium on Multiple-Valued Logic (ISMVL)*, 2020, pp. 265–270. doi: 10.1109/ISMVL49045.2020.00011.

[45] R. J. Wilson, *Introduction to Graph Theory*. USA: John Wiley &amp; Sons, Inc., 1986.

[46] C. Y. Lee, “Representation of switching circuits by binary-decision programs,” *The Bell System Technical Journal*, vol. 38, no. 4, pp. 985–999, 1959, doi: 10.1002/j.1538-7305.1959.tb01585.x.

[47] Akers, “Binary Decision Diagrams,” *IEEE Transactions on Computers*, vol. C–27, no. 6, pp. 509–516, 1978, doi: 10.1109/TC.1978.1675141.

[48] S. Nagayama, A. Mishchenko, T. Sasao, and J. Butler, “Exact and heuristic minimization of the average path length in decision diagrams,” *Journal of Multiple-Valued Logic and Soft Computing*, vol. 11, Aug. 2005.

[49] C. E. Shannon, “A symbolic analysis of relay and switching circuits,” *Transactions of the American Institute of Electrical Engineers*, vol. 57, no. 12, pp. 713–723, 1938, doi: 10.1109/T-AIEE.1938.5057767.

[50] V. Dvorák, “Bounds on Size of Decision Diagrams.,” *J. UCS*, vol. 3, pp. 2–22, Apr. 1997.

[51] J. E. Newton and D. E. Verna, “A Theoretical and Numerical Analysis of the Worst-Case Size of Reduced Ordered Binary Decision Diagrams,” *ACM Transactions on Computational Logic*, 2018, [Online]. Available: https://hal.archives-ouvertes.fr/hal-01880774

[52] B. Bollig and I. Wegener, “Improving the variable ordering of OBDDs is NP-complete,” *IEEE Transactions on Computers*, vol. 45, no. 9, pp. 993–1002, 1996, doi: 10.1109/12.537122.

[53] F. A. Aloul, I. L. Markov, and K. A. Sakallah, “Faster SAT and Smaller BDDs via Common Function Structure,” in *University of Michigan*, 2001, pp. 443–448.

[54] F. Aloul, I. Markov, and K. Sakallah, “FORCE: A Fast and Easy-to-Implement Variable-Ordering Heuristic,” *IEEE Great Lakes Symposium on VLSI*, Aug. 2003, doi: 10.1145/764808.764839.

[55] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, pp. 42–47. doi: 10.1109/ICCAD.1993.580029.

[56] M. Fujita, Y. Matsunaga, and T. Kakuda, “On variable ordering of binary decision diagrams for the application of multi-level logic synthesis,” in *Proceedings of the European Conference on Design Automation.*, 1991, pp. 50–54. doi: 10.1109/EDAC.1991.206358.

[57] N. Ishiura, H. Sawada, and S. Yajima, “Minimization of binary decision diagrams based on exchanges of variables,” in *1991 IEEE International Conference on Computer-Aided Design Digest of Technical Papers*, 1991, pp. 472–475. doi: 10.1109/ICCAD.1991.185307.

[58] S. Shirinzadeh, M. Soeken, and R. Drechsler, “Multi-objective BDD optimization for RRAM based circuit design,” *Formal Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2016*, May 2016, doi: 10.1109/DDECS.2016.7482461.

[59] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli, “Logic verification using binary decision diagrams in a logic synthesis environment,” pp. 6–9, 1988, doi: 10.1109/ICCAD.1988.122451.

[60] K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient implementation of a BDD package,” in *27th ACM/IEEE Design Automation Conference*, 1990, pp. 40–45. doi: 10.1109/DAC.1990.114826.

[61] K. Karplus, “REPRESENTING BOOLEAN FUNCTIONS WITH IF-THEN-ELSE DAGs,” University of California at Santa Cruz, Santa Cruz, 1988.

[62] A. Mishchenko, “Introduction to zero-suppressed decision diagrams,” *Synthesis Lectures on Digital Circuits and Systems*, vol. 45, Aug. 2001.

[63] R. I. Bahar *et al.*, “Algebraic decision diagrams and their applications,” in *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*, 1993, pp. 188–191. doi: 10.1109/ICCAD.1993.580054.

[64] Y.-T. Lai and S. Sastry, “Edge-valued binary decision for multi-level hierarchical verification,” *[1992] Proceedings 29th ACM/IEEE Design Automation Conference*, pp. 608–613, 1992, doi: 10.1109/DAC.1992.227813.

[65] A. Shrestha and L. Xing, “A logarithmic binary decision diagram-based method for multistate system analysis,” *IEEE Trans Reliab*, vol. 57, no. 4, pp. 595–606, 2008, doi: 10.1109/TR.2008.2006038.

[66] X. Zang, D. Wang, H. Sun, and K. S. Trivedi, “A BDD-Based Algorithm for Analysis of Multistate Systems with Multistate Components,” *IEEE Transactions on Computers*, vol. 52, no. 12, pp. 1608–1618, Dec. 2003, doi: 10.1109/TC.2003.1252856.

[67] D. Bugaychenko, “On application of multi-rooted binary decision diagrams to probabilistic model checking,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7148 LNCS, pp. 104–118, 2012, doi: 10.1007/978-3-642-27940-9_8.

[68] J. Lind-Nielsen, “BuDDy – A Binary Decision Diagram Package.” Accessed: Apr. 26, 2024. [Online]. Available: https://github.com/jgcoded/BuDDy

[69] F. Somenzi, “CUDD: CU Decision Diagram Package.” Accessed: Apr. 26, 2024. [Online]. Available: https://github.com/vscosta/cudd

[70] T. van Dijk, *Sylvan: multi-core decision diagrams*, no. 16–398. University of Twente, 2016. doi: 10.3990/1.9789036541602.

[71] A. Walker, “cudd: Bindings to the CUDD binary decision diagrams library.” Accessed: Apr. 27, 2024. [Online]. Available: https://hackage.haskell.org/package/cudd

[72] I. Filippidis, “dd: Library of decision diagrams and algorithms on them, in pure Python, as well as Cython bindings to CUDD, Sylvan, and BuDDy.” Accessed: Apr. 27, 2024. [Online]. Available: https://pypi.org/project/dd/0.4.2/

[73] A. Vahidi, “JDD: a pure Java BDD and Z-BDD library.” Accessed: Apr. 27, 2024. [Online]. Available: https://bitbucket.org/vahidi/jdd

[74] M. Research, “DecisionDiagrams.” Accessed: Apr. 27, 2024. [Online]. Available: https://github.com/microsoft/DecisionDiagrams

[75] M. Mrena, M. Kvassay, and E. Zaitseva, “TeDDy: Templated Decision Diagram Library,” *SoftwareX*, p. to appear, 2024.

[76] M. Mrena, “TeDDy: Templated Decision Diagram library.” 2024.

[77] S. Minato, N. Ishiura, and S. Yajima, “Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean function Manipulation.,” in *27th ACM/IEEE Design Automation Conference. Proceedings 1990,* Jul. 1990, pp. 52–57. doi: 10.1109/DAC.1990.114828.

[78] D. M. Miller and R. Drechsler, “Implementing a multiple-valued decision diagram package,” in *Proceedings. 1998 28th IEEE International Symposium on Multiple-Valued Logic (Cat. No.98CB36138),* 1998, pp. 52–57. doi: 10.1109/ISMVL.1998.679287.

[79] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[80] T. Krkoška, “Softvérová knižnica pre manipuláciu s binárnymi rozhodovacími diagramami,” University of Žilina, 2019.

[81] M. Kvassay, E. Zaitseva, V. Levashenko, and J. Kostolny, “Binary Decision Diagrams in reliability analysis of standard system structures,” in *2016 International Conference on Information and Digital Technologies (IDT),* 2016, pp. 164–172. doi: 10.1109/DT.2016.7557168.

[82] M. Kvassay, E. Zaitseva, V. Levashenko, and J. Kostolny, “Multi-valued Decision Diagrams for k-Out-of-n Three-State Systems,” in *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL),* 2017, pp. 260–265. doi: 10.1109/ISMVL.2017.38.

[83] M. Kvassay, E. Zaitseva, P. Sedlacek, and P. Rusnak, “Multi-valued Decision Diagrams in Reliability Analysis of Consecutive k-out-of-(2k-1) Systems,” in *2021 IEEE 51st International Symposium on Multiple-Valued Logic (ISMVL),* 2021, pp. 81–86. doi: 10.1109/ISMVL51352.2021.00023.

[84] R. E. Bryant, “Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams,” *ACM Comput. Surv.,* vol. 24, no. 3, pp. 293–318, Sep. 1992, doi: 10.1145/136035.136043.

[85] D. M. Miller and R. Drechsler, “On the construction of multiple-valued decision diagrams,” *Proceedings 32nd IEEE International Symposium on Multiple- Valued Logic,* pp. 245–253, 2002, doi: 10.1109/ISMVL.2002.1011095.

[86] L. Xing and Y. S. Dai, “A new decision-diagram-based method for efficient analysis on multistate systems,” *IEEE Trans Dependable Secure Comput,* vol. 6, no. 3, pp. 161–174, 2009, doi: 10.1109/TDSC.2007.70244.

[87] “The GNU Multiple Precision Arithmetic Library.”

[88] G. Hutton, *Programming in Haskell,* 2nd ed. USA: Cambridge University Press, 2016.

[89] M. Kvassay, P. Rusnak, E. Zaitseva, and J. Kostolny, “Application of logic differential calculus and binary decision diagrams in detection of minimal cut vectors,” *Proceedings of the 29th European Safety and Reliability Conference, ESREL 2019*, pp. 802–809, 2020, doi: 10.3850/978-981-11-2724-3_0447-CD.

[90] A. Shrestha, L. Xing, and Y. Dai, “Decision Diagram Based Methods and Complexity Analysis for Multi-State Systems,” *IEEE Trans Reliab*, vol. 59, no. 1, pp. 145–161, 2010, doi: 10.1109/TR.2009.2034946.

[91] P. Fišer, “Collection of Digital Design Benchmarks.” 1991. [Online]. Available: https://ddd.fit.cvut.cz/www/prj/Benchmarks/

[92] M. Mrena and M. Kvassay, “Comparison of Left Fold and Tree Fold Strategies in Creation of Binary Decision Diagrams,” in *2021 International Conference on Information and Digital Technologies (IDT)*, 2021, pp. 341–352. doi: 10.1109/IDT52577.2021.9497593.

[93] K. McElvain, “IWLS’93 Benchmark Set : Version 4.0,” *Distributed as a part of IWLS’93 benchmark set*, 1993.

[94] M. Mrena, P. Sedlacek, and M. Kvassay, “Linear Fold and Tree Fold in Creation of Binary Decision Diagrams of Standard Benchmarks,” in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2021, pp. 1120–1125. doi: 10.1109/IDAACS53288.2021.9660940.

[95] M. Mrena, M. Kvassay, and S. Stankevich, “Dynamic Binary Decision Diagram Creation Using an Extended Apply Algorithm,” in *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, 2023, pp. 513–518. doi: 10.1109/IDAACS58523.2023.10348726.

[96] M. Mrena, M. Kvassay, and S. Czapp, “Single and Series of Multi-valued Decision Diagrams in Representation of Structure Function,” in *Lecture Notes in Networks and Systems*, 2022, pp. 176–185. doi: 10.1007/978-3-031-06746-4_17.

[97] M. Mrena and M. Kvassay, “Experimental Analysis of Decision Diagrams Used to Represent Structure Functions of Series-Parallel Multi-State Systems,” in *ESREL 2022: Proceedings of the 32nd European Safety and Reliability Conference*, Singapore: RESEARCH PUBLISHING, 2022, pp. 1707–1714. doi: 10.3850/978-981-18-5183-4_R29-09-653-cd.

[98] M. Mrena and M. Kvassay, “Efficient Computation of Logic Derivatives Using Multi-valued Decision Diagrams,” in *ISMVL 2024 IEEE International Symposium on Multiple-Valued Logic*, 2024, p. to appear.

[99] M. A. Thornton and V. S. S. Nair, "Efficient calculation of spectral coefficients and their applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 11, pp. 1328–1341, 1995, doi: 10.1109/43.469660.

[100] S. S. Skiena, *The Algorithm Design Manual*, 2nd ed. Springer Publishing Company, Incorporated, 2008.

[101] M. Mrena and M. Kvassay, "Experimental Survey of Algorithms for the Calculation of Node Traversal Probabilities in Multi-valued Decision Diagrams," in *Reliability Engineering and Computational Intelligence for Complex Systems: Design, Analysis and Evaluation* , C. van Gulijk, E. Zaitseva, and M. Kvassay, Eds., Cham: Springer Nature Switzerland, 2023, pp. 3–20. doi: 10.1007/978-3-031-40997-4_1.

[102] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[103] A. V Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*. in Addison-Wesley series in computer science and information processing. Addison-Wesley, 1983.

[104] S. Du, R. Kang, Z. Zeng, and E. Zio, "Time-dependent reliability assessment of a distributed generation system based on multi-valued decision diagrams and Markov processes," in *Safety and Reliability – Theory and Applications*, CRC Press, Jun. 2017. doi: 10.1201/9781315210469-413.

[105] C. Bauer, A. Frink, and R. Kreckel, "Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language," *J Symb Comput*, vol. 33, no. 1, pp. 1–12, 2002, doi: https://doi.org/10.1006/jsco.2001.0494.

[106] P. Rusnak, J. Rabcan, M. Kvassay, and V. G. Levashenko, "Time-Dependent Reliability Analysis Based on Structure Function and Logic Differential Calculus," in *International Conference on Dependability of Computer Systems*, 2018.

[107] "The R Project for Statistical Computing." 2024. Accessed: Apr. 28, 2024. [Online]. Available: https://www.r-project.org/

[108] "ggplot2: system for declaratively creating graphics." [Online]. Available: https://ggplot2.tidyverse.org/

[109] Steinbach Bernd and Posthoff Christian, *Boolean Differential Calculus*. Morgan & Claypool, 2017. doi: 10.2200/S00766ED1V01Y201704DCS052.

[110] M. Kvassay and E. Zaitseva, "Topological analysis of multi-state systems based on direct partial logic derivatives," *Springer Series in Reliability Engineering*, vol. 0, no. 9783319634227, pp. 265 – 281, 2018, doi: 10.1007/978-3-319-63423-4_14.

[111] M. Mrena, P. Sedlacek, and M. Kvassay, "Linear Fold and Tree Fold in Creation of Binary Decision Diagrams of Standard Benchmarks," in *Proceedings of the 11th IEEE International Conference on Intelligent Data Acquisition and Advanced*

Computing Systems: Technology and Applications, IDAACS 2021, 2021. doi: 10.1109/IDAACS53288.2021.9660940.

[112] M. Mrena and M. Kvassay, “Comparison of Left Fold and Tree Fold Strategies in Creation of Binary Decision Diagrams,” in *International Conference on Information and Digital Technologies 2021, IDT 2021*, 2021. doi: 10.1109/IDT52577.2021.9497593.

Appendices

Appendix A – Pseudocodes

Appendix A contains pseudocodes of existing algorithms referenced from the main sections of the thesis. The presented pseudocodes were introduced in the referenced literature and are not an original contribution of this thesis. All the pseudocodes were adjusted to conform to the style, notation, and conventions used in this thesis. Otherwise, they contain little to no modification of the original ideas, though, in some algorithms, we present a simple straightforward generalization of an algorithm originally proposed just for BDDs. Finally, considering the implementation aspects, the pseudocodes assume, for simplicity, that the diagrams use the default order of variables i.e., that for an internal node A it holds that $\text{INDEX}(A) = \text{LEVEL}(A)$.

```
procedure CREATETERMINALNODE(value)
    if CONTAINS(terminalTable, value) then
        return LOOKUP(terminalTable, value)
    else
        node ← TERMINALNODE(value)              ▷ Create new terminal node
        PUT(terminalTable, value, node)
        return node
    end if
end procedure
```

Alg. A.1 Factory function for the creation of terminal nodes commonly used in decision diagram packages

```
procedure CREATEINTERNALNODE(index, sons)
    if ISREDUNDANT(sons) then
        return HEAD(sons)                       ▷ Return the first element
    else if CONTAINS(uniqueTable, (index, sons)) then
        return LOOKUP(uniqueTable, (index, sons))
    else
        node ← INTERNALNODE(index, sons)        ▷ Create new internal node
        PUT(uniqueTable, (index, sons), node)
        return node
    end if
end procedure
```

Alg. A.2 Factory functions for the creation of internal nodes commonly used in decision diagram packages

```
procedure FromVector(vector)
    stack ← MakeEmptyStack
    j ← 0
    while j < Size(vector) do
        sons ← MakeTuple($m_n$)                          ▷ Create tuple of $m_n$ elements
        for k = 0 to $m_n$ do
            sons[k] ← CreateTerminalNode(vector[j])
            Increment(j)
        end for
        node ← CreateInternalNode(n, sons)
        Push(stack, (node, n))
        ShrinkStack(stack)
    end while
    (root, _) ← Peek(stack)
    return MDD(root)
end procedure

procedure ShrinkStack(stack)
    loop
        (node, i) ← Peek(stack)
        if i = 1 then                                    ▷ Peeked node is the root node
            return
        end if
        k ← 0
        count ← 0
        repeat
            (_, j) ← Peek(stack, k)                      ▷ Peek kth element from the top
            if j = i then
                count ← count + 1
            end if
            k ← k + 1
        until k < Size(stack) ∧ i = j
        if count < $m_i$ − 1 then
            return
        end if
        sons ← MakeTuple($m_i$ − 1)
        for k = 0 to $m_i$ − 1 do
            (son, _) ← Pop(stack)
            sons[k] ← son
        end for
        node ← CreateInternalNode(i − 1, sons)
        Push(stack, (node, i − 1))
    end loop
end procedure
```

Alg. A.3 From-vector – static algorithm for the creation of MDD from truth vector [80]

```
procedure APPLY(left, right, $\odot$)
    root ← APPLYSTEP(ROOT(left), ROOT(right), $\odot$)
    return MDD(root)
end procedure

procedure APPLYSTEP(left, right, $\odot$)
    if CONTAINS(applyCache, (left, right)) then
        return LOOKUP(applyCache, (left, right))
    end if
    if ISTERMINAL(left) $\wedge$ ISTERMINAL(right) then
        node ← CREATETERMINALNODE(VALUE(left) $\odot$ VALUE(right))
    else if ISABSORBINGTTERMINAL($\odot$, left) $\vee$ ISABSORBINGTTERMINAL($\odot$, right) then
        node ← CREATETERMINALNODE(ABSORBINGELEMENT($\odot$))
    else
        $i_{lhs}$ ← LEVEL(left)
        $i_{rhs}$ ← LEVEL(right)
        $i$ ← min($i_{lhs}$, $i_{rhs}$)
        sons ← MAKETUPLE($m_i$)
        for $k = 0$ to $m_i$ do
            if $i_{lhs} < i_{rhs}$ then
                sons[k] ← APPLYSTEP(SON(left, k), right)
            else
                sons[k] ← APPLYSTEP(left, SON(right, k))
            end if
        end for
        node ← CREATEINTERNALNODE(i, sons)
    end if
    PUT(applyCache, (left, right), node)
    return node
end procedure
```

Alg. A.4 Entry point ant the recursive step of the apply algorithm [11], [12]

```
procedure SatisfyCount(diagram, value)
    root ← Root(diagram)
    iroot ← Index(root)
    diff ← DomainProduct(1, iroot)
    count ← diff ∗ SatisfyCountStep(root, value)
    return count
end procedure

procedure SatisfyCountStep(node, value)
    if IsTerminal(node) ∧ Value(node) = j then
        return 1
    end if
    if IsTerminal(node) ∧ Value(node) ≠ j then
        return 0
    end if
    if Contains(memo, node) then
        return Lookup(memo, node)
    end if
    count ← 0
    i ← Level(node)
    for k = 0 to $m_i$ do
        son ← Son(node, k)
        $i_{son}$ ← Level(son)
        sonCount ← SatisfyCountStep(son, value)
        diff ← DomainProduct(i, $i_{son}$)
        count ← diff ∗ sonCount
    end for
    Put(memo, node, count)
    return count
end procedure

procedure DomainProduct($i_1$, $i_2$)
    product ← 1
    i ← $i_1$
    while $i < i_2$ do
        product ← product ∗ $m_i$
        i ← i + 1
    end while
    return product
end procedure
```

Alg. A.5 Entry point and the recursive step of the satisfy-count algorithm [11]

```
procedure COFACTOR(diagram, i, a)
    root ← ROOT(diagram)
    if ISTERMINAL(root) then
        return diagram
    else if INDEX(root) = i then
        newRoot ← SON(root, a)
        return MDD(newRoot)
    else
        newRoot ← COFACTORSTEP(root, i, a)
        return MDD(newRoot)
    end if
end procedure

procedure COFACTORSTEP(node, i, a)
    if CONTAINS(memo, node) then
        return LOOKUP(memo, node)
    end if
    if ISTERMINAL(node) then
        return node
    end if
    if INDEX(node) = i then
        return SON(node, a)
    end if
    if INDEX(node) > i then
        return node
    end if
    j ← INDEX(node)
    sons ← MAKETUPLE($m_j$)
    for k = 0 to $m_j$ do
        oldSon ← SON(node, k)
        sons[k] ← COFACTORSTEP(oldSon, i, a)
    end for
    newNode ← CREATEINTERNALNODE(j, sons)
    PUT(memo, node, newNode)
    return newNode
end procedure
```

Alg. A.6 Entry point and the recursive step of the cofactor algorithm [11]

```
procedure CalculateNTPPostStep($node$, $values$)
    if IsTerminal($node$) then
        $value \leftarrow$ Value($node$)
        if Contains($values$, $value$) then
            return 1.0
        else
            return 0.0
        end if
    end if
    if Contains($memo$) then
        return Lookup($memo$, $node$)
    end if
    $i \leftarrow$ Index($node$)
    $resut \leftarrow 0.0$
    for $k = 0$ to $m_i$ do
        $son \leftarrow$ Son($node$, $k$)
        $sonProb \leftarrow$ CalculateNTPPostStep($son$, $values$)
        $result \leftarrow result + sonProb * p_{i,k}$
    end for
    Put($memo$, $node$, $result$)
    return $result$
end procedure
```

Alg. A.7 Post-order NTP calculation algorithm (bottom-up approach) [10]

```
procedure CALCULATENTPLEVEL(diagram)
    root ← ROOT(diagram)
    stacks ← ARRAY($n + 2$)        ▷ Create an array of stacks (queues or lists can also be used)
    stackIndex ← INDEX(root)
    PUSH(stacks[stackIndex], root)
    PUT(memo, root, 1.0)
    while stackIndex $< n + i$ do
        stack ← stacks[stackIndex]
        while ISNOTEMPTY(stack) do
            node ← POP(stack)
            i ← INDEX(node)
            if ISINTERNAL(node) then
                nodeNTP ← LOOKUP(memo, node)
                for $k = 0$ to $m_i$ do
                    son ← SON(node, k)
                    if CONTAINS(memo, son) then
                        sonNTP ← LOOKUP(memo, son)
                    else
                        sonNTP ← 0.0
                        $i_{son}$ ← INDEX(son)
                        PUSH(stacks[ison], son)
                    end ifs
                    sonNTP ← sonNTP + nodeNTP $* p_{i,k}$
                    PUT(memo, son, sonNTP)
                end for
            end if
        end while
        while stackIndex $< n + 1 \wedge$ ISEMPTY(stacks[stackIndex]) do
            stackIndex ← stackIndex + 1
        end while
    end while
    return memo
end procedure
```

Alg. A.8 Level-order NTP calculation algorithm (top-down approach) [10]

Appendix B – List of Publications

1. M. Mrena and M. Kvassay, “Comparison of left fold and tree fold strategies in creation of binary decision diagrams,” in *2021 International Conference on Information and Digital Technologies (IDT)*, 2021, pp. 341–352.
2. M. Mrena, P. Sedlacek, and M. Mrena, “Linear fold and tree fold in creation of binary decision diagrams of standard benchmarks,” in *2021 11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 2, 2021, pp. 1120–1125.
3. P. Rusnak and M. Mrena, “Time dependent reliability analysis of the data storage system based on the structure function and logic differential calculus,” in *Reliability Engineering and Computational Intelligence*, C. van Gulijk and E. Zaitseva, Eds. Cham: Springer International Publishing, 2021, pp. 179–198
4. M. Mrena, M. Kvassay, and S. Czapp, “Single and series of multi-valued decision diagrams in representation of structure function,” in *Lecture Notes in Networks and Systems, vol. 484 LNNS*, 2022, pp. 176–185.
5. M. Mrena and M. Kvassay, “Experimental analysis of decision diagrams used to represent structure functions of series-parallel multi-state systems,” in *Proceedings of the 32nd European Safety and Reliability Conference (ESREL 2022)*, 2022, pp. 1707–1714.
6. M. Mrena, M. Varga, and M. Kvassay, “Experimental comparison of array-based and linked-based list implementations,” in *2022 IEEE 16th International Scientific Conference on Informatics (Informatics)*, 2022, pp. 231–238.
7. M. Mrena and M. Kvassay, “Comparison of single MDD and series of MDDs in the representation of structure function of series-parallel MSS,” in *2022 IEEE 16th International Scientific Conference on Informatics (Informatics)*, 2022, pp. 225–230.
8. P. Galcik, M. Mrena, L. Piatrikova, and S. Stankevich, “Advanced priority queues in the optics clustering algorithm,” in *2023 International Conference on Information and Digital Technologies (IDT)*, 2023, pp. 257–266.
9. M. Mrena and M. Kvassay, “Experimental survey of algorithms for the calculation of node traversal probabilities in multi-valued decision diagrams,” in *Reliability Engineering and Computational Intelligence for Complex Systems: Design, Analysis and Evaluation*, C. van Gulijk, E. Zaitseva, and M. Kvassay, Eds. Cham: Springer Nature Switzerland, 2023, pp. 3–20.
10. M. Mrena, M. Kvassay, and S. Stankevich, “Dynamic binary decision diagram creation using an extended apply algorithm,” in *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2023, pp. 513–518.

11. M. Mrena and M. Kvassay, “Generating monotone Boolean functions using Hasse diagram,” in *2023 IEEE 12th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, 2023, pp. 793–797.
12. M. Mrena and M. Kvassay, “Efficient computation of logic derivatives using multi-valued decision diagrams,” in *ISMVL 2024 IEEE International Symposium on Multiple-Valued Logic*, 2024, to appear.
13. M. Mrena, M. Kvassay, and E. Zaitseva, “Teddy: Templated decision diagram library,” *SoftwareX*, to appear, 2024.