Multiple Valued Decision Diagrams in the Diagnosis of IT Systems

András Vörös
vorike@gmail.com

András Pataricza
pataric@mit.bme.hu

Model-Based Assessment of IT Services and Components
BME DMIS - IBM FA research project

June 27, 2009

Abstract

Integrated system level diagnosis has been a well articulated set of disciplines for the last two decades. Although, interpreted largely for binary fault, error and failure domains, integrated system level diagnosis principle provides approaches that are by far not restricted to its original domain of inception, electromechanical and embedded (largely) safety-critical systems. However, the diagnostic approaches utilized in today's generic IT service and system management do not draw on this established knowledge.
As a recent research shows [2], a feasible direction of system level diagnosis and impact analysis of heterogeneous, distributed IT systems is that of qualitative static error propagation modeling and analysis by constraint satisfaction approaches. Here, a) faults errors and failures are modeled not in a binary way but still in a qualitative manner with small sets of discrete values; and b) the dynamic dependability related behavior of component inputs and outputs is categorized into discrete values of a finite set, called syndromes, thereby giving arise to component level error propagation descriptions that are effectively relations in a mathematical sense. From all these relations and dependencies we are able to compute the possible diagnostic states of the system as the solutions to a constraint satisfaction problem.
The time needed for the reaction is a critical point in recent on-line systems as the availability highly depends on it. The task to keep the services up is a challenging task. Reaction time can be reduced by faster incoming data processing.
In my work I tried to find a good representation for the diagnostic states. As the number of these states can be high, memory efficiency is an important requirement. Additionally, it must provide fast access and manipulation abilities to enable fast reaction to the incoming information. For these purposes I decided to use Multiple Valued Decision Diagrams.
I developed my own Java based MDD implementation and the required interfaces, and I integrated it to the test environment developed by FTSRG research group.
I tested the storing capabilities of this solution on a model of a virtual environment. The state space consisted of 240 nodes, which means that with the help of a Java VM using 1 GByte memory we can manage up to 28 000 similar services simultaneously. The time required to process the incoming information is about 300 times faster then the formerly used PROLOG based implementation.
Additionally, this MDD implementation is able to solve the CSP problem, comparing to the formerly used PROLOG solver it produced similar runtime results.

Table of Contents

1. Introduction 5
2. Data structures 7
   2.1 Binary Decision Diagrams 7
   2.1.1 Basic Definitions 7
   2.1.2 Reduced Ordered Decision Diagrams 8
   2.2 Multiple Valued Decision Diagrams 9
   2.2.1 Multiple Valued Functions 9
   2.2.2 Multiple Valued Decision Diagrams 10
   2.2.3 Construction of MDD-s 11
   2.2.4 Multiple valued function decomposition and depiction 12
   2.2.5 Data structures, functions, building an MDD 15
3. MDD implementation 16
   3.1 Implementation with adjacent level interchange 16
   3.1.1 Drawbacks of using operator nodes 16
   3.1.2 Main criteria 16
   3.2 Implementation with recursive functions, operations 16
   3.2.1 Recursive operations 17
   3.3 Testing 19
   3.3.1 General purposes 19
   3.3.2 Performance tests 19
4. Supporting Diagnosis with MDD-s 24
   4.1 Diagnosis and MDD 24
   4.1.1 Representing diagnostic systems 24
   4.2 Integrating MDD to a test environment 25
   4.3 Constraint solving and MDD 25
   4.4 Example system's constraint solving measures 27
   4.5 Representing the solution set 28
   4.6 Supporting the diagnosis 28
5. Conclusion, further work 29
6. References 30

1. Introduction

Due to the technological development, the appearance of extremely powerful hardware, highly versatile software and super fast networks, all connected to each other worldwide; the complexity of IT services is also growing. These architectures, the provided services and still the environment are dynamic. However, the demand to have the control over these systems remained the same; we have to have a full sight to the system with the ability to react very quickly to the changes.

It is a challenging task to keep the services up and to fulfill the reliability and availability requirements.

Modern IT service management systems provide new generation means to observe and to react to the changes of the managed systems. An IBM Tivoli Monitoring server can gain information from many information sources, as it uses agents in the observed computers and servers, and they provide real-time information of the system's health state. Nowadays hundreds of servers including database-, web- and application servers can be observed by a single monitoring application. However, not only large amount of server, but many types of servers can be observed simultaneously. This huge amount of information, which contains both useful and unnecessary information, should be examined in a fast way. Modern IT systems have to provide 0,9999 or higher availability, so that the speed of the reaction is a critical point.

How do recent service management solutions work? At first, the information collector agents are deployed to the system components. We can decide which components and parameters will be under examinations. After any configuration modifications we have to reconfigure the monitoring system too.

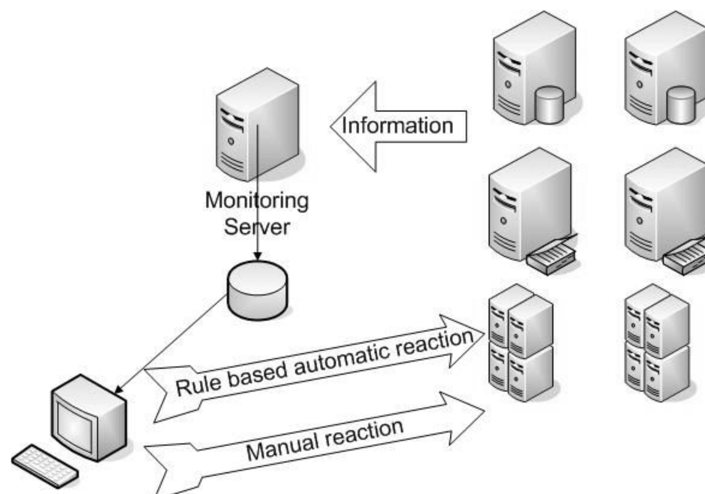

Figure 1. Modern IT service management

How can we react to the diagnostic events? We can make rules in the monitoring system that will be executed if a triggering event happened. This is a simple way of rule based reconfiguration. Another possibility is that if the rule based reaction doesn't work, the operator, who sits at the monitoring application, estimates the situation, and manually reacts. This is probably not a fast and efficient way of managing large systems as it highly depends on personal knowledge and experience. These ad-hoc controlling and diagnosis doesn't support the provisioning of large systems.

Nowadays, when model based development is a widespread paradigm, the idea of model based diagnosis came naturally [1]. Derivating the system model from the architecture and from the components' model is a good way of treating the system together. The importance of this approach is that unlike to the former event based reaction it provides sight to the whole system, and we can decide not only from a trigger event but we can take into account the information from other components. In this case the systems' state space become available, both the components' states and the observations.

As the reaction is a critical point, my aim was to advice a proper data structure which supports the diagnosis. The requirements are the following:

- able to cope with large amount of information, many observations
- fast, as we can have a large amount of observations simultaneously
- memory efficient, as the state space can be extremely large
- support hierarchical modeling
- support refinement possibilities

In my work I developed a solution which is independent from the underlying model; it serves as a fast reasoning system which can be suited to any kind of model. In the first sections I introduce the main expectations against the data structures and the basics of the decision diagrams, both binary and multiple-valued cases. In the third section I show my implementation and the characteristics of it. I show an example in the last section, where I introduce the integration of my program to an example system developed by the FTSRG research group just to show how it works on an example model.

2. Data structures

There were two main requirements against the used data structure:

- space effective representation
- speed consideration, it should provide fast access and manipulation ability

The speed of the diagnostic process heavily depends on the data structures used for storing the solution set.

In this paper I try to find a proper data representation to make the diagnostic process faster. I propose to use decision diagrams as they offer efficient storage possibilities of binary and multi-valued functions with the exploitation of the redundancy occurring in them, and they provide fast access and manipulation abilities. Deciding whether a solution is in a solution set, it needs $O(n)$ time, where n it's the number of variables depicted in the representation.
In my work I examined the efficiency of building the data structure from solution vectors and the other possibility, when MDD based constraint solving leads to the solution set, without any other programs. Both alternatives proved their usability.
In the following sub-sections I introduce in chronological order the two main types of decision diagrams.

2.1 Binary Decision Diagrams

Decision diagrams are widely used in all fields of mathematics and technology as they provide a compact and easy to understand representation of functions. There are many types of decision diagrams: for example, Binary Decision Diagrams (BDD-s), Zero Suppressed BDD-s [3], Structurally Synthesized BDD-s [4], or Multiple Valued Decision Diagrams (MDD-s; see later).
For various reasons, today BDD-s are the most predominantly used flavor. As the MDD formalism heavily builds on the established notions of and technique dealing with BDD-s, this section summarizes the knowledge on BDD-s that can be treated as a prerequisite from our point of view.

2.1.1 Basic Definitions

Definition: Binary Decision Diagram is a rooted directed acyclic graph $G=(V,E)$ with vertex set V containing two types of vertices, non-terminal and terminal vertices. A non-terminal vertex v has as a label a variable $value(v)=\{x_1,x_2...x_n\}$ and two children $low(v)$ and $high(v)\in V$. A terminal vertex v is labeled with a value: $value(v)\in\{0,1\}$ and has no outgoing edge.

A BDD is called ***ordered BDD*** if in every path variables have the same ordering and none of them appears more than once in a single path.

A BDD is called ***reduced ordered BDD*** (ROBDD) under these two circumstances:
Uniqueness: no two distinct nodes u and v have the same variable name and low- and high-successor $value(v)=value(u)$, $low(v)=low(u)$, $high(v)=high(u)$ implies $u=v$

Non-redundant tests: no variable node v has identical low- and high-successor $low(v)\neq high(v)$

Binary Decision Diagrams actually encode Boolean functions: $f:B^n\mapsto B$. Due to their characteristic properties, ROBDD-s are appropriate to visualize Boolean functions in a compact form; also, they are a very space-efficient form for storing Boolean functions.

2.1.2 Reduced Ordered Decision Diagrams

ROBDD is a canonical representation of an *n*-variable Boolean function for a fixed order of input variables. ROBDD-s are widely used in formal verification and synthesis algorithms. However, the permutation of the variable order very often gives different ROBBD for the same Boolean function. Hence, in many cases the size of the ROBDD can be significantly different depending on the variable order. Still, ROBDDs usually have a compact size, but there are problems known to have exponential ROBDD representation. (Size of an ROBDD is understood as the number of nodes in the diagram graph.)

It is known to be an NP-hard problem to find ROBDD with the least number of nodes for a given Boolean function. However, there are some heuristics that generally find a ROBDD with acceptable size for a given function.

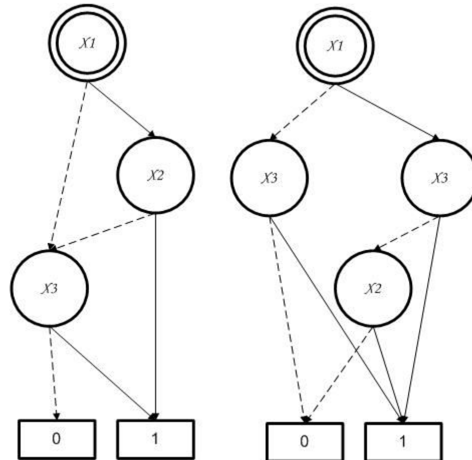

Figure 2. Effect of variable ordering in ROBDD representation

As ROBDD-s are nothing more than data structures, it is possible to reorder the variables in an ROBDD "on-the-fly", naturally with some processing time penalty. Currently, the most popular dynamic reordering technique is the so-called sifting algorithm. In this algorithm variables of a binary function are ordered at first in some initial order: $(x_1, x_2 ... x_n)$. In the next stage each variable is moved through the current variable ordering using swaps. The sifting algorithm tries to find a good variable ordering of a ROBDD by successive analyzing each variable, starting from the initial order. The investigated variable is moved through the whole ordering. Finally, the variable is moved to its optimal position. Because each variable is moved only once, in the sifting procedure only n^2 swaps are needed.[5] This algorithm was integrated with spectral analysis of the function using the first order Walsh coefficients in some other work. The efficiency of this algorithm depends on the variable ordering before the processing. [8]

The algorithm can be accelerated by using the fact that there can be symmetric variables in the represented Boolean function. Two variables are said to be symmetric, if "swapping" them in the Boolean function results in the same Boolean function. Symmetry properties can be used to efficiently construct good variable orders for ROBDD's using modified gradual improvement heuristics. The crucial point is to locate symmetric variables side by side and to treat them as a fixed block. This technique is motivated by the following three facts:

1) The exchange of two symmetric variables does not change the size of the ROBDD, because the function remains the same.
2) The size of the ROBDD of any totally symmetric function (all variables are pair wise symmetric): $f : \{0,1\}^n \mapsto \{0,1\}$ is $O(n^2)$.

3) The value of a function which is symmetric in some variables $\left\{x_{i_1}, x_{i_2}, ..., x_{i_q}\right\}$ does not depend on the exact assignment of these variables but only on their weight $\sum_{j=1}^{q} x_{i_j}$.

Using the first fact, the heuristics can skip over the exchange of symmetric variables and so the run time decreases. However, the resulting ROBDD sizes will be the same. The second and third facts lead to the special class of variable orders of the technique, i.e., variable orders where the symmetric variables are located side by side, and then we can treat them as a fixed block. Hereby we receive a modification of sifting: the symmetric sifting algorithm, which sifts symmetric groups simultaneously. Regular sifting usually puts symmetric variables together in the order, but the symmetric groups tend to be in suboptimal positions. The suboptimal solutions result from the fact that regular sifting is unable to recognize that the variables of a symmetric group have a strong attraction to each other and should be sifted together. When a variable of a symmetric group is sifted by regular sifting, it is likely to return to its initial position due to the attraction of the other variables of the group. [9]

However, there are other measures that can be subject to optimization for a ROBDD. These can incorporate average path lengths (APL) [6][7], the maximal width of the decision diagram, number of the edges, and so on. As usual, optimality for one measure does not imply optimality regarding another one.

2.2 Multiple Valued Decision Diagrams

Heavily drawing on the previous one, this section reviews the basic definitions, properties and operations of Multiple Valued Decision Diagrams.

2.2.1 Multiple Valued Functions

A multi-valued variable X_i can take on values from a finite set $V_i \in \left\{v_0, v_1, v_2, ..., v_{|V|-1}\right\}$. Because each symbolic value v_i can be associated with a unique integer *i*, without the loss of generality we can restrict our treatment to multi-valued variables with integer values: $V_i \in \left\{0,1,...,|V|-1\right\}$. A multi-valued function is formally: $f:\left\{V_1, V_2, ..., V_j\right\} \mapsto V_k$; an *n*-variable *m*-valued function is a mapping: $f:\left\{0,1,...,m-1\right\}^n \rightarrow \left\{0,1,...,m-1\right\}$.

It can be shown that the *{MIN, MAX, literals}* set of functions is functionally complete for *m*-valued functions, where the respective functions are defined the following way:

MIN: $V_i \in \left\{0,1,...,|V|-1\right\}, V_i \in \left\{0,1,...,|V|-1\right\} : MIN\left(V_i, V_j\right) = if\left(V_i < V_j\right) then\, V_i\, else\, V_j$

It is easy to see that considering binary variables MIN is equal to AND:

x\y	0	1
0	0	0
1	0	1

MAX: $V_i \in \{0,1,...,|V|-1\}, V_i \in \{0,1,...,|V|-1\} : MAX(V_i, V_j) = if(V_i > V_j)\, then\, V_i\, else\, V_j$

It is easy to see that considering binary variables MAX is equal to OR:

x\y	0	1
0	0	1
1	1	1

Literals: $J := \left\{ \overset{1}{V_1}, \overset{2}{V_2}, ..., \overset{m-1}{V_{m-1}} \right\}$ set of literal operators defined by:

$$\overset{i}{V_i} = \begin{cases} m-1, V_i = i \\ 0, otherwise \end{cases}$$

A simple example with binary variables:

x	$\overset{0}{x}$	$\overset{1}{x}$
0	1	0
1	0	1

It is easy to see that considering binary variables $\overset{0}{x}$ is equal to negation.

Numerous data structures are known for representing multi-valued functions, such as for example generalized **Karnaugh** maps or **Multiple Valued Decision Diagrams**. In this work, we focus on Multiple Valued Decision Diagrams.

2.2.2 Multiple Valued Decision Diagrams

Definition: a multi-valued (or multiple decision diagram) is a rooted directed acyclic graph $G = (V, E)$ with vertex set V containing two types of vertices, non-terminal and terminal vertices. A non-terminal vertex labeled by variable index $value(v) = \{x_1, x_2 ... x_n\}$ and has up to *p* labeled outgoing edges where *p* is the multiplicity of variable X_i , edges from a single node may have up to *p*-1 values as their label.

An MDD is called an ***ordered MDD*** if in every root to terminal vertex path variables have the same ordering and all of them appears only once in a single path.

Formally: each node has outgoing edges: $v \to (e_0, e_1 ... e_i) \in E$, $i \le p-1$, where e_i is the *i*-th edge going out from the node *v*, each edge has a label *l*: $e_i \to l_i$, $l_i \in \{0,1,...,p-1\}^j$, $j \le p-1$ (for ROMDD only), $\forall i, j : l_i \cap l_j = \emptyset$, $l_0 \cup l_1 \cup .... \cup l_i = \{0,1,...,p-1\}$.

This means that more than one edge from node *v* may point to the same ending node. A MDD has up to *p* terminal nodes representing function values. However, usually we use another approach.

It is possible to represent multiple valued functions with more than one MDD. In this case we assign to each output value a single MDD with terminal nodes 0 and 1, in this (sub)MDD we represent the functions resulting the given output. This may result in less complicated MDD-s, but multiple ones for a single function.

An MDD is called *a **reduced ordered MDD (ROMDD)*** under these two circumstances:

- Uniqueness: no two distinct nodes *u* and *v* have the same variable name and successors: let $e^u{}_i$ be the node *u*'s *i*-th edge with labeling $l^u{}_i$ and $successor\left(e^u{}_i\right)$ is the node pointed by $e^u{}_i$. In this case: $value(v) = value(u)$, $\forall i : successor\left(e^u{}_i\right) = successor\left(e^v{}_i\right)$, $\forall i : l^u{}_i = l^v{}_i$ implies $u = v$
- Non-redundant tests: no variable node *v* has identical successors: $\exists i, j : successor\left(e_i\right) \neq successor\left(e_j\right)$

In the following we show that the general properties of MDD-s and their applicability for storing functions or relations as was with BDD-s for the Boolean domain.

The construction of MDD-s is similar to that of BDD-s. The construction of MDD-s is based on the generalized Shannon expansion. However, instead of the *ITE* operator we use the *CASE* operator:
Definition: $CASE\left(x, y_0, y_1 ... y_{p-1}\right) = y_i \;\; if \;\; x = i$.

Definition: Generalized Shannon expansion:

$$f\left(x_1, x_2, ..., x_n\right) = \overset{0}{x_i} \bullet f\big|_{x_i=0} + \overset{1}{x_i} \bullet f\big|_{x_i=1} + ... + \overset{m-1}{x_i} \bullet f\big|_{x_i=m-1}$$

where $f\big|_{x_i=j}$ are the cofactors of f defined by $f\big|_{x_i=j} = f\left(x_1, x_2, x_{i-1}, j, x_{i+1} ..., x_n\right)$ for all $i \in \{1, 2, ..., n\}$ $j \in M$, and "+","•" denote the multiple-valued operations MAX and MIN correspondently. Note that the literal operator could be defined in a more general way:
$\overset{I}{x} = \begin{cases} m-1 & if \; x \in I, \\ 0 & otherwise \end{cases}$, where *I* is a subset of the domain.

2.2.3 Construction of MDD-s

The construction of MDD-s is similar to the binary case. It is important to mention, that while MDD-s are the generalization of BDD-s, every MDD can be represented by a BDD. The core idea is that we code the multiple valued variables with binary ones. Each MDD variable with multiplicity *p* needs $\lceil \log_2 p \rceil$ binary variables for encoding. Usually we don't need all binary values; in this case "don't care" variables appear in the following sense.
Let us assume that we encode a three valued variable. It can be coded with two bits:

$$0 \to 00; \;\; 1 \to 01; \;\; 2 \to 10, 11 = 1_; \;\; \text{where} \;\; _ = dont \; care$$

To avoid the effect of "don't care"-s it is recommended to represent each output value with a separate MDD. Why is it important? Using this approach it can be avoided that the number of solutions are increased by the "don't care"-s. However it doesn't mean in the one-MDD representation the real increase of the number of solutions just the effect that with a traversing we get back some solutions twice, but this redundancy can be removed after decoding.
How can we avoid the effect of "don't care"-s? In one solution-one MDD representation a single MDD represents just one function output. So we can use exact codes instead of using don't care-s: $0 \to 00; \;\; 1 \to 01; \;\; 2 \to 11$, and code 10 goes constantly to terminal node 0, which means this is not the part of that output. Why don't use this method in a multiple output MDD? Because in this case we use 0 as an output value not for signing that this path doesn't correspond to that output value.

Binary Decision Diagrams are the most commonly used tool for storing MDD-s. MDD-s can be represented in their own without a sub-BDD representation. The main difference between a BDD representation and MDD representation is the number of edges starting from a node. In MDD representation the list of edges which belongs to a single node contains not only two edges. The main disadvantage of using BDD-s for storing MDD-s is that we have to create nodes instead of bigger edge

lists. For example using BDD instead of an MDD with variables having a domain 4, we have to create 2 variable levels for each MDD variables; in this case we will use up to 3 nodes instead of 1. However after the reduction some of them will be eliminated, but in a common case not all.
The problem with coding MDD-s in BDD-s is that the efficiency highly depends on the coding.

During the course of this work, I have examined many BDD packages and finally I have found that for now it is more practical to use and MDD implementation instead of representing MDD-s with BDD-s. The main difference between an MDD and a BDD implementation is the node structure. In a BDD each node has two descendants, in an MDD implementation they have more.

2.2.4 Multiple valued function decomposition and depiction

In this subsection I provide an example for building an MDD from an operator-based representation of a multi valued function.
Let us treat the function with the following generalized **Karnaugh** map:
$f:\{x,y,z\}\mapsto\{0,1,2\}, \quad x,y,z\in\{0,1,2\}$

$x \setminus y,z$	00	01	02	10	11	12	20	21	22
0	0	1	2	0	1	0	0	0	0
1	1	1	2	0	1	0	0	1	1
2	2	2	2	2	2	2	0	1	2

The function is the following with the *MIN, MAX, literal* operators:

$$f(x,y,z)=1\bullet\overset{0}{x}\bullet\overset{0}{y}\bullet\overset{1}{z}+2\bullet\overset{0}{x}\bullet\overset{0}{y}\bullet\overset{2}{z}+1\bullet\overset{0}{x}\bullet\overset{1}{y}\bullet\overset{1}{z}+$$
$$+1\bullet\overset{1}{x}\bullet\overset{0}{y}\bullet\overset{0}{z}+1\bullet\overset{1}{x}\bullet\overset{0}{y}\bullet\overset{1}{z}+2\bullet\overset{1}{x}\bullet\overset{0}{y}\bullet\overset{2}{z}+1\bullet\overset{1}{x}\bullet\overset{1}{y}\bullet\overset{1}{z}+1\bullet\overset{1}{x}\bullet\overset{2}{y}\bullet\overset{1}{z}+1\bullet\overset{1}{x}\bullet\overset{2}{y}\bullet\overset{2}{z}+$$
$$+2\bullet\overset{2}{x}\bullet\overset{0}{y}\bullet\overset{0}{z}+2\bullet\overset{2}{x}\bullet\overset{0}{y}\bullet\overset{1}{z}+2\bullet\overset{2}{x}\bullet\overset{0}{y}\bullet\overset{2}{z}+2\bullet\overset{2}{x}\bullet\overset{1}{y}\bullet\overset{0}{z}+2\bullet\overset{2}{x}\bullet\overset{1}{y}\bullet\overset{1}{z}+2\bullet\overset{2}{x}\bullet\overset{1}{y}\bullet\overset{2}{z}+1\bullet\overset{2}{x}\bullet\overset{2}{y}\bullet\overset{1}{z}+2\bullet\overset{2}{x}\bullet\overset{2}{y}\bullet\overset{2}{z}$$

After generalized Shannon decomposition:

$$f(x,y,z)=\overset{0}{y}\bullet(x+z)+\overset{1}{y}\bullet\left(2\bullet\overset{2}{x}+1\bullet\overset{1}{z}\right)+\overset{2}{y}\bullet x\bullet z$$

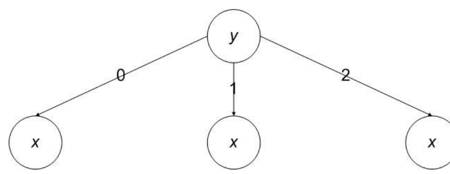

Figure 3. Branches from variable y

$\overset{0}{y}$:

$x \setminus y,z$	00	01	02
0	0	1	2
1	1	1	2
2	2	2	2

$$f(x,z)=x+z=\overset{0}{x}\bullet z+\overset{1}{x}\bullet\left(1\bullet\overset{0,1}{z}+\overset{2}{z}\right)+\overset{2}{x}$$

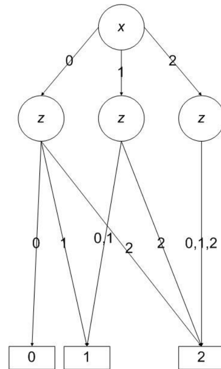

Figure 4. Sub-MDD of y=0 branch

$\overset{1}{y}$:

x \ y,z	10	11	12
0	0	1	0
1	0	1	0
2	2	2	2

$$f\left(x,z\right) = 2\bullet\overset{2}{x} + 1\bullet\overset{1}{z}$$

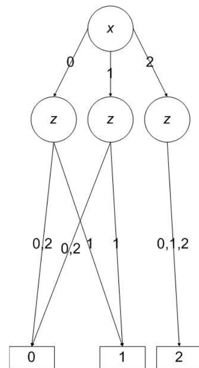

Figure 5. Sub-MDD of y=1 branch

$\overset{2}{y}$:

x \ y,z	20	21	22
0	0	0	0
1	0	1	1
2	0	1	2

$$f\left(x,z\right) = x\bullet z = 1\bullet\overset{1}{x}\bullet y + \overset{2}{x}\bullet y$$

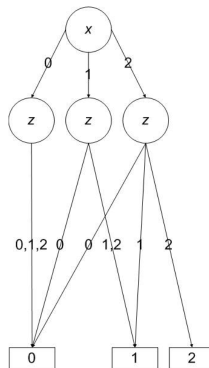

Figure 6. Sub-MDD of y=2 branch

After the generalized Shannon decomposition we are able to depict the function as an MDD:

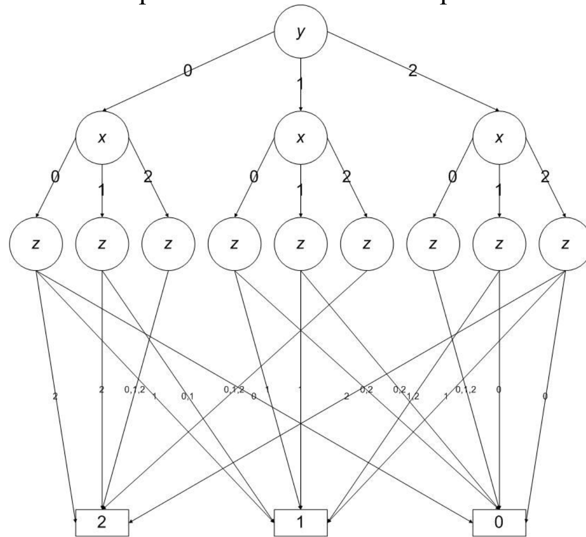

Figure 7. MDD without reduction

During the construction or after it we can reduce the MDD to an ROMDD.

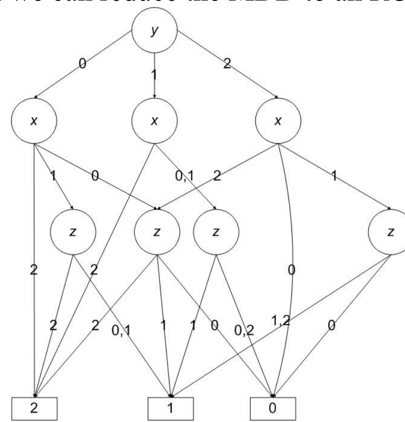

Figure 8. ROMDD

This ROMDD can be represented by 3 ROMDD-s corresponding to each output value of the function:

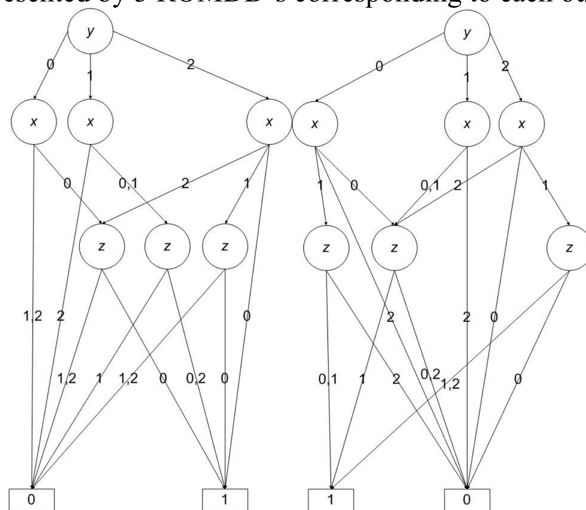

Figure 9. ROMDD-s for output values 0 and 1

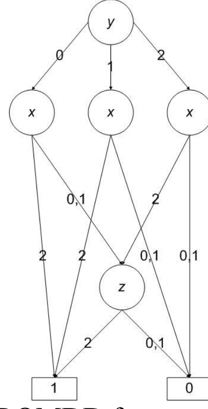

Figure 10. ROMDD for output value 2

In the following the term MDD refers to Reduced Ordered MDD.

2.2.5 Data structures, functions, building an MDD

We have to store nodes and their relationships efficiently. Each edge can be represented as a single pointer to a node. They are stored in arrays inside the node structure. In a node we also have to register which edge points to which node, the label of the nodes, and the value of the variable represented by the node.

RESTRICT operator. An assignment to a multi-valued variable restricts the possible values it can assume.

$$RESTRICT(f, x_i, I):$$

$$f(x_1, x_2, ..., x_n) \mapsto f(x_1, x_2, ...x_{i-1}, x_i = i_1, x_{i+1}, ..., x_n), ..., f(x_1, x_2, ...x_{i-1}, x_i = i_k, x_{i+1}, ..., x_n)$$

$$I \in \{i_1, ..., i_k\}$$

The **RESTRICT** operation can be expressed with the generalized Shannon decomposition:

$$RESTRICT(f, x_i, I): f \mapsto \overset{I}{x_i} \bullet f|_{x_i=0}, \overset{I}{x_i} \bullet f|_{x_i=1}, ..., \overset{I}{x_i} \bullet f|_{x_i=m-1}$$

APPLY operator. We can apply multiple valued operations to MDD-s. We use the generalized Shannon decomposition:

$$f_1(x_1, x_2, ..., x_n)\, op\, f_2(x_1, x_2, ..., x_n) = \overset{0}{x_i} \bullet \left(f_1|_{x_i=0}\, op\, f_2|_{x_i=0}\right) + \overset{1}{x_i} \bullet \left(f_1|_{x_i=1}\, op\, f_2|_{x_i=1}\right) + ... + \overset{m-1}{x_i} \bullet \left(f_1|_{x_i=m-1}\, op\, f_2|_{x_i=m-1}\right)$$

By recursion the operands will go down to the constant level where we can apply them directly.

Satisfiability related functions. The usual approach is that if a multi-valued functions value is not 0, then it is treated as “true”, else it is “false”:

SATCOUNT: $SATCOUNT(f(X)) = |\forall X : f(X) \neq 0|$

ALLSAT: $ALLSAT(f(X)) = \forall X : f(X) \neq 0$

How can we build an MDD from a set of variable valuation vectors over multi-valued variables? Since we use this representation to store a CSP solution set we use only two terminal nodes: 0 and 1. 0 means that the given solution is not a part of the solution set; paths ending in the terminal node 1 are the feasible solutions. Our task is to store them efficiently.

Here efficiency can mean:

- Compact representation
- Possibility of efficient manipulation
- Fast access to stored information

3. MDD implementation

The basics of my work were the previously developed c programming language based MDD implementation, published by the Victoria University [10] . However, I developed my own java based implementation, to fulfill the portability and other requirements. I used in the first implementation the same algorithms as the c package.

3.1 Implementation with adjacent level interchange

I implemented the algorithm used in the package of Victoria University. The logical functions were implemented as the formerly introduced [10] adjacent level interchange algorithm. This means still increased memory because of the unnecessarily created operator nodes.
I left the algorithm the same with some adjustment to the specialties of the java language.
These were:

- usage of object oriented data structures
- using java specific (generic) pre-written structures instead of the raw structures used in c program language
- dividing the sources from the interface, more structured code
- some additional checking functions

3.1.1 Drawbacks of using operator nodes

As the authors of the [10] mentioned in their paper, there is another way of processing operations in MDD-s, with recursive functions. They didn't publish their recursive program, but they mentioned that it is faster and it uses less memory.
Not only memory usage is the drawback of the usage of operator nodes. We use processor time not only to create, but then to process these nodes.
The other insufficiency of the algorithm was that it was developed just for representational reasons, there wasn't any cleaning and garbage collecting function implemented. As we wanted to use it in our diagnostic process for large solution sets and for real-life problems, I had to eliminate these problems.

3.1.2 Main criteria

As a usual requirement, this program should be fast and memory efficient. After some tests, I could decide that the problem with the adjacent level interchange is the increased size of memory allocation during the operations. As long as the size of the state space doesn't grow linearly during the operations, it is not a good idea to create a single operator node to each sub-function. The growing size of the nodes because of the java language increased the final size of the program 3 times, the required memory peak during the operations has fallen drastically, and the result of it was also the increased speed of the execution of operations.
Another important memory saving was the development of reference counting. It is a complex task; it needs many reference computations during the operations, as long as many functions can handle nodes and restructure the diagram. As many nodes can be created during the functions, many can become unnecessarily, and this reference handling can avoid the unnecessary node aggregation in the unique table, which can make the program slower. The c program I used formerly doesn't contain any kind of garbage collection, so, after the hash tables are filled up with nodes, it slows down the program noticeably, and these nodes consumes a very large amount of memory, unnecessarily.
As the formerly mentioned MDD package has some algorithmic and functional drawbacks, I developed my own algorithm implementation.

3.2 Implementation with recursive functions, operations

I have rewritten the program in a new way. I implemented the operations with recursive functions and I implemented real time garbage collection too.

3.2.1 Recursive operations

The former algorithm firstly created operator nodes, and then exchanged them to MDD nodes. This required increased memory as each sub operation, function was presented by an operator node. So each node was pre-processed; and instead of each node we created and used two nodes indeed.
As in the early stage turned out, this is not the fastest way of computing logic operations.
That’s why I started to develop my own implementation.
Main point in my operation realization:

- recursive functions
- real time garbage collection
- usage of the formerly presented hash tables with increased efficiency of the hash function

Recursive functions:

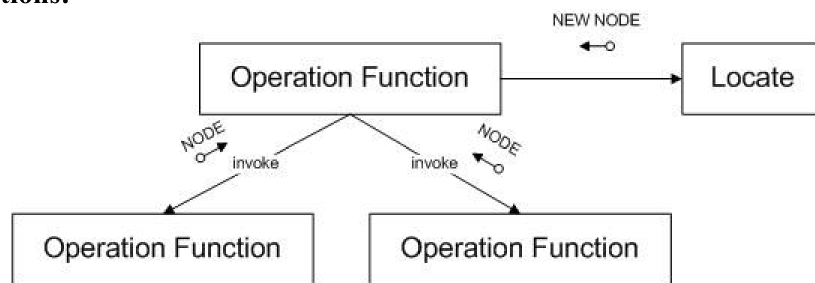

Figure 11. Operation of recursive functions

As it can be seen in the figure above, we create for each sub-function a single node only after the computation of the whole sub-function.
It provides solutions in a single depth-first traversal, which algorithm doesn’t need as much memory as the creation of operator nodes.

Real time garbage collection:
I don’t use cyclic negation in my program. As the formerly presented algorithm works, it provided 7% less nodes, but 12,5% more place for a single node, so all in all, it just increased the real size of the decision diagram.

In my program I register in each node the number of references pointing to it. It also helped to increase the speed of the algorithm, because I don’t have to use cleaning functions after all operations. Instead I maintain the references during the processing. This maintenance was needed in 2 kinds of functions:

- recursive operations
- locating functions

Reference maintenance needs the propagation of reference changes to lower level nodes in the case of redundancy elimination. I created a reference change propagation function, which solves this problem.
Another important situation is when due to the constructional rules, we have to eliminate nodes. In this case we also have to propagate this reference decrease to the lower levels, which is also solved by this reference propagating function.
I don’t use the java in-built garbage collector to free the nodes which are not used more; I created a stack for these nodes so we are able to reuse them, we don’t have to use the Java garbage collector.
The algorithm uses this stack only when it is not possible to use immediately the node being freed. If during the recursive calls a node becomes free, we use it up it to the next generated node without any use of this storage functionality. This seemed to be the fastest way to compose the new MDD-s.

MDDNode structure:

MDDNode
-id : long -value : int -flag : int -refcount : int -hash : int -edgelist[] : MDDNode
#getid() : long #setid() #incref() #decref() #getval() : int #setval() #getchild() : MDDNode #setchild() #isTerminal() : bool #inithash() +hashCode() : int +equals() : bool

Figure 12. class MDDNode

id: This variable is unique, each node has a different one. I used this variable in the generation of the hash values of the nodes. As far this *id* is counted in the MDD class, we can easily gain the number of nodes required during the operations. This is useful when doing memory consumption and complexity measures.
value: This variable defines the variable level of the node. If the node is a Terminal node, the value of the node is 0 and the function *isTerminal()* returns true.
flag: I used this just for checking purposes, it is not necessary.
refcount: It is used to register the references in the node. If the *refcount* value is 0, then we need this node no more, so we can free it up and use it up for other nodes. Function *incref()* increments the value, *decref()* decrements the value of it. As the program uses the same node structure for every kind of node, we can easily reuse them for storing other nodes. This was also a big advantage of the avoidance of the usage of the operator nodes.
hash: This is the hash value of the node, it is used when the program places the node into the unique hash table. At the construction of the node we can set this value with the *inithash()* function. This is the return value of the *HashCode()* function. It was created because I tested the program with many kinds of hash functions. I used not only the id for this purpose, but the hash value of the children nodes. I developed two efficient hash functions, they have similar efficiency. Finally I used the following hash function:

```
protected void inithash() {
   this.hash = (int)this.edgelist[0].getid();
   for(int i = 1; i < this.edgelist.length; i++){
       this.hash *= 7;
       this.hash += this.edgelist[i].getid();
   }
}
```

This function needs less computation than the other one using the child nodes' hash values. The drawback of this hash function is that if we reach the end of the domain of the long, it can cause serious problems. This can be reached not only if the domain is so big, but we use many nodes during the computation.
edgelist[]: This array is used to store the pointers of the child nodes. Unfortunately - as we don't really now the size of the domain in advance - this *edgelist* becomes a single object, stored apart from the node. It causes the increased size of a node. For example, a node from domain 4 uses up 48 bytes of memory for storing its' own data and it uses up 56 bytes for storing the *edgelist* array. So each node from this domain uses up 104 bytes of memory.
It is easy to see that it can be easily reduced if we don't use the unnecessarily created fields, but we don't gain as much free memory as much more work will be needed to check and process the MDD.

Comparing to the program in c program language, Java uses for a single node the 325% of memory of the c program, 104 bytes to 32 bytes. This difference can be decreased if we don't only want to use this program for proof-of-concept purposes. I showed in my thesis the ways of this further development.
It is worth knowing that MDD based storing means memory efficient storage capabilities only for redundant functions. If our function is a real-random function, which means little redundancy, that can cause the loss of this memory efficiency. However, the manipulation remains still easy and fast.
The equals function is used to decide whether two nodes are equal or not. It can be decided only from the *endgelist*-s without any more information; it is useful if we want to expand our MDD with more levels, then we don't have to re-compute all nodes in the unique table.
Unique structure: I used the *java.util.HashMap* [19] class to store the unique table data. This was the easiest way to store the nodes, as it can grow itself if it is needed, so I only had to pay attention to the MDD specific parts of the programming. I used this class as a set of nodes; unfortunately the *java.util.HashSet* class has the same hash table backup as the *HashMap* class, so it wouldn't mean less memory consumption. I completed this class with some collision counter procedures in order to make countable the collisions during the processing of the MDD-s. It was essential in the development of a good hash function.

3.3 Testing

3.3.1 General purposes

After the development steps we always want to make sure that the program does what we want. Testing can provide evidence of it. Additionally it is worth knowing what kind of performance properties does our program have. As I formerly mentioned, our aim was to develop a program which provides memory effective and fast solution to our state space storing requirements. All these properties were tested.
At first I tested the functionality. I created function *issolution()*, which determines whether a vector is part of the solution set in the MDD or not. Function *satcount()* determines the number of vectors contained by the MDD. The testing process consists of adding vectors to the solution set, and then I checked if all these vectors are in the MDD, and no other vector is in the MDD. I used a hash table to store the vectors I placed formerly to the MDD. After a few millions of solution vectors I checked if the number of vectors in the hash table equals the number of the vectors in the MDD. Then I checked if each vector in the hash table is in the MDD. I did this procedure for millions of vectors in many kinds of domain and vector size, and all tests were successful. It is important not only to store the vectors, but to store them properly. In an ROMDD I had to ensure the reduced property. In my program the interface of the unique table provides this property. The *_locate()* method is responsible for keeping up this property. If we want to place a node to the unique table, this function examines if the same node exists, and places the new node only if it is unique, else gives back a reference to the formerly placed node. This method is also responsible for preventing nodes to be placed, if the nodes' all children are the same. At second, I tested whether the references are kept in proper state. I created a reference checking function, which stored the references in the flag, which were computed during a depth-first traversal, and then we only had to examine the equivalence of the flags and reference counters in each node. I created a few million test cases for this test purpose, and I examined after each modification the correctness of the reference counters. It took a long time, but finally all test cases became successful. The final test ran about 2 hours long and I simulated a full solution set during this test.
For this test we used MDD-s having only two terminals, 0 and 1, so-called Shared MDD, as we wanted to depict set functions. In the following, if I refer to MDD, I mean Shared MDD.

3.3.2 Performance tests

I created a random vector generator class. Why did I need this? I wanted to test the performance of this program through a whole domain. My own random generator let me to iterate through a full domain without repetition of the elements. Another important property is that it is a pseudo-random generator:

I could repeat any test with absolutely the same circumstances. It is important when I want to compare hash functions or other measures. I wanted to reach as many nodes in the MDD as it is possible in a domain. For this purpose I had to use a generator with minimum entropy between the elements. As it is well known, raising to a power in a prime group results real random numbers. So I used the following algorithm: At first, after the computation of the size of the domain, I choose a bigger prime number. Then I have to choose a number, which can be the generator element of the modulus group. It is not evident to find a good generator element, which element generates the full domain, but after a few try I could always find a good one. So, formally:

$a^p \equiv a \bmod p$ if p is prime; I try to find an n: $a^p \equiv a \bmod p$ and $\forall n \in 0....p\text{-}1:\ a^n \neq a \bmod p$

This generator class allowed me to make more accurate performance measures, and because of the deterministic property I was able to compare the efficiency of the hash functions to each other.

Generation time examples:
1 048 576 vectors:

`VGen(4,10,7,1048583 )` – runtime: 1201 ms

16 777 216 vectors:

`VGen(4,12,7,16777259)` – runtime: 22734 ms

It can be easily seen, that it is slower than the in-built java random number generators.

3.3.2.1 Hash function testing

I tested the hash functions with the first random number generator and full vector domain ($4^{10} = 1048576$ solution vectors). During the computation the program created 12 148 186 nodes in various levels. With the formerly presented hash function we get 4 945 438 collisions during the construction. I tested with many other hash functions, but here I present only one more:

```
int h = (int) this.edgelist[this.edgelist.length-1].hashCode();
this.hash = (int)this.edgelist[0].hashCode();
for(int i = 1; i < this.edgelist.length; i++){
  this.hash *= 7;
  h *= 137;
  this.hash += this.edgelist[i].hashCode();
  h += this.edgelist[this.edgelist.length-1-i].hashCode();
}
h = 32467 ^ h;
this.hash *= ~h;
```

This hash function produced 4 906 216 collisions during the construction.
As their deviation in collisions is about 0,8%, they have similar runtimes, but because of the less computation needs the first hash function I introduced needed about 3% less time to run.
The hash function used by the c program produces 7 189 744 collisions, which is 46,5% higher than the number of collisions produced by my own development.
I tried also the Java *HashCodeBuilder* class to increase the efficiency, but it produced 7 101 647 collisions for this test, which is 44,75% higher than my solution.
I tried some of the advised hash functions from Thomas Wang [21], but they didn't offer a better performance, however I used some of the ideas I found there.

3.3.2.2 Global performance test

I tested the runtime, the differential time, the memory usage and a number of the nodes during the construction. The x axis shows the number of solutions added to the decision diagram, each step in the scale means 1024 more added solutions. In the following I show my runtime results.

I tested four parameters of my program:

Runtime:

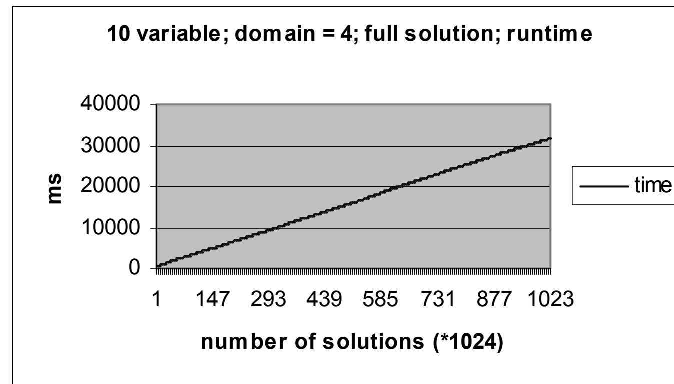

Figure 13. Runtime results in a 10 variable domain

The time to add new solutions to the diagram increased nearly linearly. As long as there is small redundancy between the vectors, it takes always the same time to construct the next MDD. This is because of the behavior of MAX operation. If we want to expand the represented solution set with one single solution, then at first we have to compute the single solution in a way I presented in the former sections, then we have to compute the MAX of this vector and the MDD containing the other solutions. As this vector means a single path to the root, with as many nodes as many variables are in the MDD, we recursively compute the MAX of each node and a node of the solution MDD. This leads to a linear runtime as the nodes we process during the operation remains in the same order of magnitude: we always compute at most two times of the number of variables, which means in the example at most 20 node manipulations per solution (we used 10 variables).
During the computation the program created and used 12 148 186 different nodes, this is 15,8% more than the number of nodes which must be created to represent the 1 048 576 solution vectors.
This test proved that this data structure is applicable for storing purposes, as it can store more than a million solution vectors. For bigger systems this can be also enough as we can reduce the solution space with the hierarchical approach I show in the following sections.
In the following figure I show a runtime result for a bigger domain:

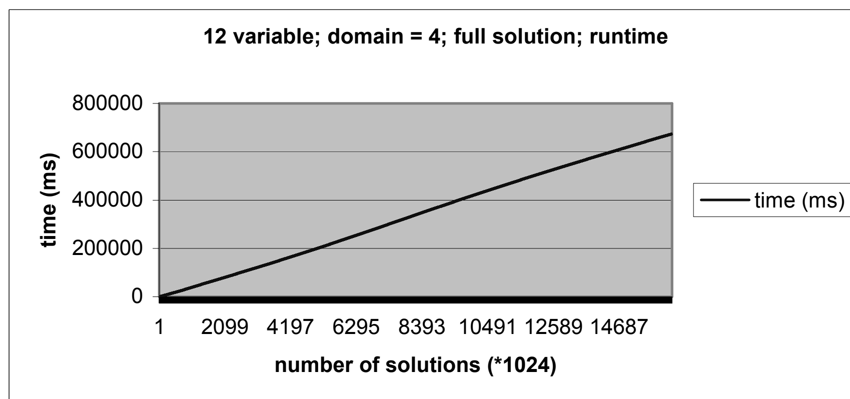

Figure 14. Runtime results in a 12 variable domain

As it can be seen, the time required to build the full solution set in this case was about 20.4 times higher than it was needed for the smaller domain. This increased time comes not only from the fact that we built up a 16 times bigger domain, but we used 20% more variable for it. Runtime results increased linearly, which is the important factor in the building of the solution set of the diagnostic state.

Differential time and memory consumption:

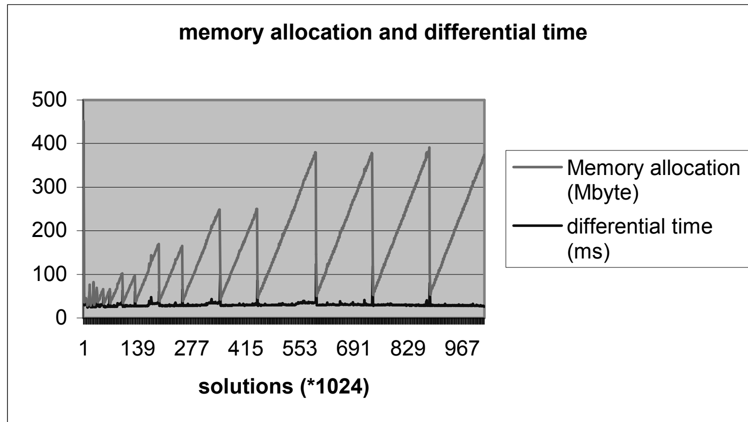

Figure 15. Memory allocation and differential time results

I examined the size of the allocated memory and the differential time together. I mean differential time the time needed to give 1024 more solutions to the solution set. There are some peaks in the diagram of differential time, then the memory consumption decreases, so these peaks show when the garbage collector works; a major garbage collection happens periodically. There are many smaller peaks, they occur when a minor garbage collection happens; these can not decrease the memory consumption spectacularly.
I examined the memory consumption of the 12 variable MDD; despite the fact that it means a 16 times bigger domain, the top of the memory consumption was just 25% higher than in the smaller case. This could happen because the MDD uses just the small part of the memory depicted in the figure above; the main part of the memory usage is caused by the variables and functions in the memory; they can be cleaned by the garbage collector.

Number of nodes

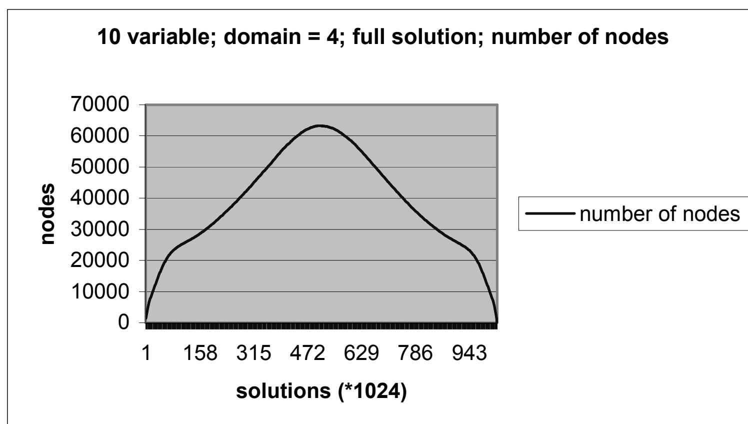

Figure 16. Number of nodes in the dependence of solution number (10 variables)

I tried to reach the maximum number of nodes in an MDD, containing 10 variables from domain 4. I tried the java in-built random number generator and my random number generator too. My generator was about 1-2% more efficient in this context.
In the next figure I show the same diagram with a bigger, 12 variable domain. The program could easily cope with 415 061 nodes.

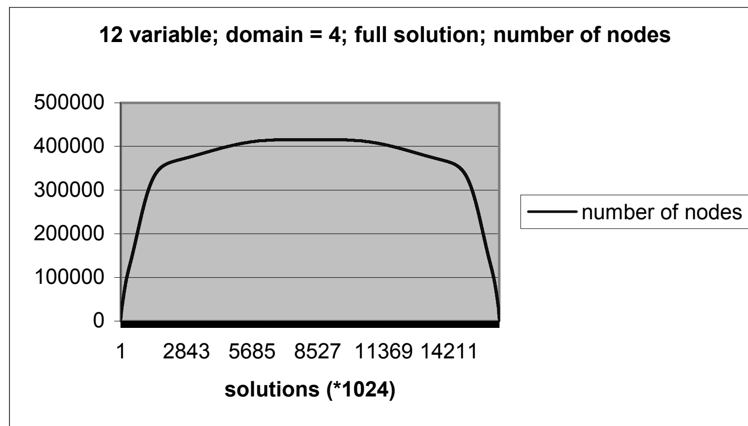

Figure 17. Number of nodes in the dependence of solution number (12 variables)

I tested still bigger domains to find the limit. It turned out that the program can process up to 7 million nodes, which is a huge amount; it is enough for our diagnostic purposes.

4. Supporting Diagnosis with MDD-s

Computing the MDD from a given set of solution is feasible. But why do we spend time to do it?
MDD-s turned to be a proper data structures to be the means to store the actual state of the diagnostic process. As I formerly examined [18] it can provide information for the further directions of diagnosis. In the following I describe the integration to the diagnostic system.
Since there are efficient algorithms for solving CSP-s [15][16], the production of the solution set is feasible, so we only have to store them in a way to exploit the redundancy and information included in it.

4.1 Diagnosis and MDD

Modeling and representing large diagnostic systems is a widely known and examined problem. There are many approaches to make diagnostic process faster and more accurate. In the following I show two representations.

4.1.1 Representing diagnostic systems

We can choose a proper representation to our system, which suits to our needs. But we have to consider some parameters of the system. For systems consisting of many smaller components, the trivial approach is the hierarchical, and for the smaller components we can use the flat approach. However there might be reasons to use flat representation, as if we have the capacity to store large models, we can profit from the speed of flat representation as we don’t have to propagate our observations and the incoming information into the component levels, and we have a full sight to the whole system.

4.1.1.1 Flat representation

This is the easiest way of representing systems, but it is efficient only for small systems. This means a single level representation; we don’t divide the system into smaller parts. The main advantage of this representation is that we can simultaneously observe all variables and components. In this case, we can use a single MDD for storing the solution set of the diagnostic system. As it is showed in the former sections, our MDD can easily store up to a few millions of solution vectors. However, if we exceed this significantly, the speed will decrease and there may be performance problems. So, at first we have to examine the solution size to decide whether this representation is the proper one.

4.1.1.2 Hierarchical representation

We don’t use a single level representation; instead we go recursively to lower levels. We have to define the interface of each component, so that we can represent the dependencies. It can be used for top-down or bottom-up diagnosis too; the direction of the diagnosis is determined by only the direction of the information flow.

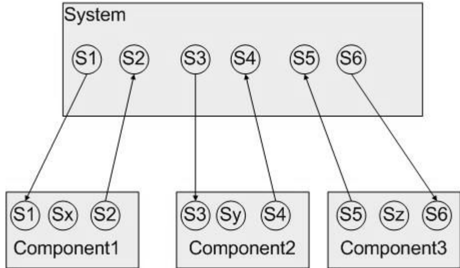

Figure 18. Hierarchical variable representation

This approach is better in large systems, because we don’t have to examine the full state space, only a small part of it. This can lead to faster data processing and faster conclusions.

Representing systems hierarchically needs smaller MDD-s, which decreases the memory consumption of the process. For example, the system above consists of 4 MDD-s, one for the System, and 3 for the Components. Each Component has some common variables with the System MDD.
This representation supports both bottom-up and top-down approaches. We only have to decide from which direction the observations are coming. For example, IT service unavailability is a top level observation, which has to be propagated to the lower levels to decide which component's failure caused the problem. However, the observation can be a failure of a component. In this case, to decide which service will then be unavailable, we have to propagate this observation to the upper levels. This is a more complex task as we have to constitute the MAX or MIN of various MDD- s.

4.2 Integrating MDD to a test environment

I integrated my Java MDD package to the diagnostic environment developed by the BME MIT FTSRG research group, especially **Imre Kocsis** and **Dániel Tóth**. The diagnostic process includes many steps.
My results are the following:

- the substitution of the prolog propagator with and MDD based constraint solver
- representing the solution set in an MDD
- observation propagation during the diagnostic process

These constraints were developed by **Imre Kocsis**. I converted these Prolog constraints into MDD constraints, so that I could solve them with my MDD program. The demo system contained 41 variables and 18 constraints, each constraint described an error propagation rule or a resource dependency among the variables.

4.3 Constraint solving and MDD

Comparing to the commonly used method: the domain store based constraint solving; MDD-s offer a new approach. To combine MDD based constrain stores and domain store based approaches turned out to be an efficient way of solving constraints [17]. I developed a more simple constraint solver, and I tested in our diagnostic environment.
The approach consists of two steps:

- convert constraints into MDD-s
- execute the logical operations on these constraints

I completed my MDD program with some functions which make the handling of constraints easier. These functions give a very effective means to handle constraints as small MDD-s. The algorithm is the following:

- with function *assignments()* it is easy to create one variable one value assignments
- to create more complex constraints, with MAX operator the combination of more assignments can be composed

The formerly presented approach defined two types of error propagation: through the data and through resource dependencies. In both case a variable or a combination of variables define the value of another variable. With the following logic primitives these dependencies can be efficiently represent:
$x = A \wedge y = B$:

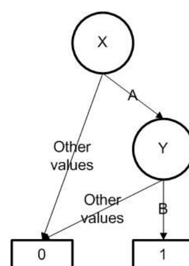

Figure 19. AND primitive

$x = A \vee y = B$:

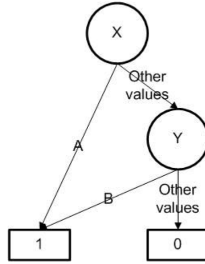

Figure 20. OR primitive

$x = A \rightarrow y = B$:

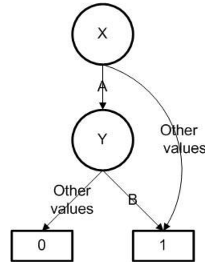

Figure 21. Implication primitive

$\neg x = A$:

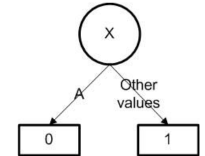

Figure 22. Negation primitive

As the error propagation can be easily defined by these primitives, error propagation can be dealt as MDD constraint on the solution set.
After the constraints are created, they can be added to the constraint store, and after it, the MDD containing the whole set of solutions is ready.
I show a simple constraint example from our diagnostic system:

Prolog code (from our example system):

```
proprule_localdisk(Faultmode, Localdisk):-
(Faultmode #= 0  #/\ Localdisk #= 0 ) #\/
(Faultmode #= 1  #/\ Localdisk #= 1 ).
```

MDD function:

```
      public MDDNode proprule_localdisk(int Faultmode,int Localdisk){
            MDDNode functroot;
            ArrayList<Integer> assignmentvect = new ArrayList<Integer>();
            for(int i = 0; i < this.varnum; i++){
            assignmentvect.add(-1);
      }
//    (Faultmode #= 0  #/\ Localdisk #= 0 ) #\/
      assignmentvect.set(Faultmode, 0);
      assignmentvect.set(Localdisk, 0);
      functroot = this.assignments(assignmentvect);

//    (Faultmode #= 1  #/\ Localdisk #= 1 ).
```

```
for(int i = 0; i < this.varnum; i++){
      assignmentvect.set(i,-1);
}
assignmentvect.set(Faultmode, 1);
assignmentvect.set(Localdisk, 1);
functroot = _DDfuncMAX(functroot,this.assignments(assignmentvect));

return functroot;
}
```

MDD based constraint solving heavily depends on the constraint compilation. As our problem consists of simple constraints, they can be easily compiled in the formerly presented way. MDD based constraint solving proved its usability for this types of problems, as it can easily used as a constraint store, and my developments added a lot to this. I especially paid attention to make the MDD package be able to store various constraints and MDD-s in the same unique table without any interference. As they are stored together, no additional computation is needed to execute the operations and to expand the store.
Former researches pointed out that MDD based constraint solving hasn't provide a good performance for *AllDiff* [17] constraints. Fortunately, in a diagnostic system, where it is not usual to have components connected to all other components, this kind of constraint is very rarely applied. All other constraints, which influence only small part of the variable space, can be efficiently compiled into an MDD constraint. In the example system, a single component was connected to at most 5 more components, which turned out to be manageable by the MDD based constraint solver.

4.4 Example system's constraint solving measures

As I didn't use domain store based constraint solving, just a constraint store, so I could measure the effects of the constraints being added to the MDD.
In the following figures I show the results:
In the first figure I show the processing time of the constraints. It is easy to see that the program spent a lot of time to add the 10th constraint to the model, which was the `proprule_virtualstorage` constraint:

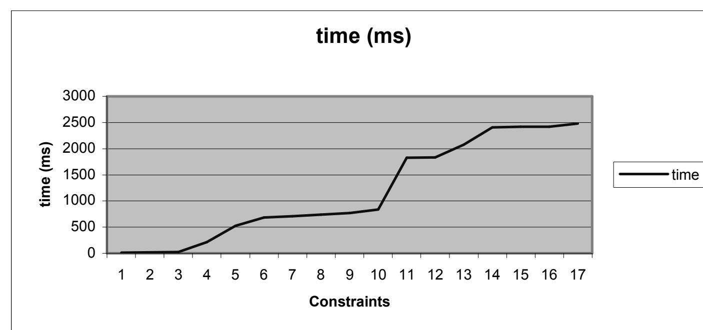

Figure 23. Runtime

To find a good order of the constraints is a subject of further researches, I examined some other orders and I got various runtime results. The main point in solving constraint satisfaction problems is to find the constraints for which the system is sensitive. Compiling these constraints as soon as possible can lead to a faster problem solving. In this problem I found 4 significant constraints, which consumed the most of the computation time. Forcing them to be computed at first led to a 10 times faster constraint solving. These constraints are called hyperactive constraints in the literature, and they are a subject of further researches.

The second figure depicts the number of the nodes contained by the MDD during the process:

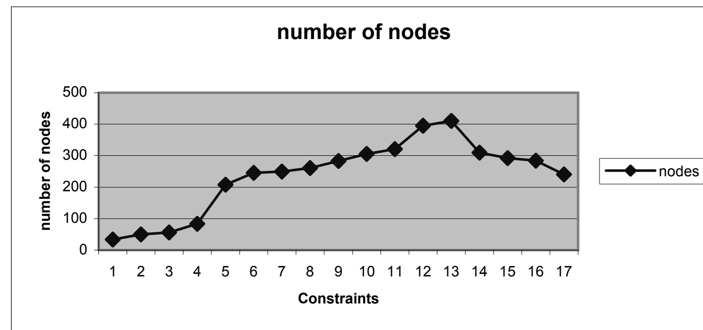

Figure 24. Number of nodes in the MDD during the computation

The number of nodes never exceeded 410, so this solution provides memory effective storage and constraint solving properties. During the constraint solving the memory consumption never exceeded the 20 MByte.

4.5 Representing the solution set

The main idea in representing the solutions in an MDD is that it provides a compact form. The solution set of our model contains 2246 solutions and we can represent these solutions on 240 nodes (including terminals, I used a 4 domain variable for each system variable). If we take into account the fact that we can store up to 7 million nodes, this can mean that we can store the solution set of 28 000 systems similar to our example system, which consists of 1148000 variables, and we can manage them simultaneously. However, if we use the flat approach, it is not worth storing them in the same MDD, because the recursion can reduce the manageability and the information retrieval speed will be slow. This problem can be solved by the hierarchical representation.
This representation can be examined from other perspectives; we are able to compute entropy of the variables easily, which can improve the efficiency of the diagnostic process [18] .

4.6 Supporting the diagnosis

MDD-s are not only applicable to store the solution set, but to compute the possible outcomes of an observation by the formerly presented restrict operation. The problem with the previous Prolog solver was that, if some diagnostic changes happened, then we had to re-compute the whole state space of the system with these additional changes. With the usage of MDD-s we don’t have to do this, we only have to propagate the observations as a constraint to the pre-computed solution MDD. It is a fast and efficient way of tracking the changes of the system. For the formerly introduced diagnostic model my MDD implementation allows 50 changes/sec tracking speed, if we would produce the new solution set with Prolog, the re-computation would take up to 7 seconds, which is 0,14 changes/sec tracking speed. For hierarchical models the advantage would be bigger.

So, our approach provided the same computational results for the offline computations, the computation of the state space where the diagnosis happens, and about 300 times faster observation compilation speed.

5. Conclusion, further work

The main aim of this work is to introduce the MDD data structure and to prove its usability in diagnostic environment. In the biggest part of my work I was developing and optimizing the Java based MDD package, I tried many functions and developments to make it faster and more memory effective. However, because of the technology, I didn't succeed in all aspects. It is easy to see, that the node structure could use less memory. If we try it in industrial environment, we can reduce the node size by 10-20% by the omission of unneeded variables, but they will be still bigger than the nodes used in the c program. Another possibility would be to restrict the domain to a pre-defined size. This can result that we don't have to use pointers in the node structure to the *edgelist* object; instead we could store them together. This would lead to more 8% reduction in node size. Or, another possible solution is to store the node as an array, which could also reduce the size because of the squandering object structure of the Java.

The speed of the algorithm was good, comparing to the c program, about 10-100 times faster depending on the density of the function and the solution number. Additionally, I couldn't try all test cases to the c program because of an internal fault, which I couldn't find in the code. This is the main drawback of the c program.

As a data representation form, MDD proved it's usability and efficiency as it could make the diagnostic process faster in a memory effective way. This approach can manage up to 10-20 000 services simultaneously, from the same complexity of the example.
The incoming information processing is still faster than the former PROLOG based solution, about 300 times, the reaction time can be reduced significantly. As far as it is a critical point in on-line diagnosis, this development may have a serious impact on the availability of the system.

A subject of further improvements is the hash table I used. It stores two pointers for each entry, which leads to more memory consumption. This can be avoided by an own implementation.
I examined the Java code with runtime profiler programs. For constraint solving the main part of the time the program spent, was spent on the MIN operation, which is good, because we only spend time on the necessary things. However, when we used Prolog to compute the solution set, and we wanted to compute the MDD from this solution vectors, the computation to create a single MDD for each vectors took up the 45% of the time. It turned out that it is not the efficient way to compute the MDD from the solution set.

The variable ordering is also a subject of further researches. As far as I know, there is no native MDD reordering programs still. This can be because to convert the MDD to BDD, then to reorder, and then to convert back needed less effort than to write a native implementation (there are many BDD variable reordering implementations). However, this is an interesting question, that whether a native MDD variable reordering algorithm can be more efficient than the other one or not.
When we used the MDD as a constraint store, it turned out, that the computation time which was needed to solve the CSP problem highly depends on the order of constraints. To find a good constraint-order is a challenging task for the future.

All together, I managed to show the justification of MDD-s in the field of system test and diagnosis.

6. References

[1] A. Pataricza.: Model based design of dependability. Dissertation for the degree of Doctor of Sciences from the Hungarian Academy of Sciences, 2006

[2] Alice X. Zheng and Irina Rish and Alina Beygelzimer: Efficient Test Selection in Active Diagnosis via Entropy Approximation; Proceedings of the 21th Annual Conference on Uncertainty in Artificial Intelligence (UAI-05); 2005

[3] S. Minato: "Zero-Suppressed BDDs and Their Applications", International Journal on Software Tools for Technology Transfer, Vol. 3, No. 2, pp. 156-170, Springer, May 2001.

[4] R. Ubar, T. Vassiljeva, J.Raik, A.Jutman, M.Tombak, A.Peder: Optimization of Structurally Synthesized BDD-s

[5] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams, " *Proc. IEEE/ACM ICCAD'98 Conf.*, pp. 42-47, 1993.

[6] Shinobu Nagayama, Tsutomu Sasao: On the Minimization of Longest Path Length for Decision Diagrams; Proc Int'l Workshop Logic Synthesis 2004

[7] SHINOBU NAGAYAMA, ALAN MISHCHENKO, TSUTOMU SASAO and JON T. BUTLER: Exact and Heuristic Minimization of the Average Path Length in Decision Diagrams; J. of Mult.-Valued Logic & Soft Computing., Vol. 11, pp. 437.465; 2005

[8] Piotr Porwik, Krzysztof Wrobel and Piotr Zaczkowski, "Some practical remarks about Binary Decision Diagram size reduction", *IEICE Electron. Express*, Vol. 3, No. 3, pp.51-57, (2006).

[9] Christoph Scholl, Dirk Möller, Paul Molitor, and Rolf Drechsler „BDD Minimization Using Symmetries", IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 18, NO. 2, FEBRUARY 1999

[10] Miller, D. M., and R. Drechsler, "Implementing a multiple-valued decision diagram package," Proc. 28th Int. Symp. on Multiple-Valued Logic, pp. 52-57, May 1998

[11] Karl S. Brace: Dept of ECE, Carnegie Mellon; Richard L. Rudell: Syinopsys, Inc.; Randal E. Bryant: School of Computer Science, Carnegie Mellon: Efficient Implementation of a BDD Package; 27th ACM/IEEE Design Automation Conference

[12] Hett, A., R. Drechsler and B. Becker, "MORE: Alternative implementation of BDD packages by multi-operand synthesis," Proc. European Design Automation Conference, pp. 164-169, 1996.

[13] Files, C.; Drechsler, R.; Perkowski, M.A.: Functional decomposition of MVL functions using multi-valued decision diagrams; Multiple-Valued Logic, 1997. Proceedings., 1997 27th International Symposium on Volume

[14] Christoph Scholl and Rolf Drechsler and Bernd Becker: Functional simulation using binary decision diagrams; In Int'l Conf. on CAD, 1997

[15] M. Carlsson, G. Ottosson, and B. Carlson: An open-ended finite domain constraint solver. Programming Languages: Implementations, Logics, and Programming, 1292:191–206, 1997.

[16] Joxan Jaffar and Michael J.Maher. Constraint logic programming: A survey. Journal of Logic Programming, 19/20:503–581, 1994.

[17] H. R. Andersen, T. Hadzic, J. N. Hooker and P. Tiedemann: A Constraint Store Based on Multivalued Decision Diagrams; Lecture Notes in Computer Science, Springer Berlin / Heidelberg

[18] Vörös András: Integrated System Level Diagnostics; Student Association Report, BME 2008

[19] java.util.HashMap: http://java.sun.com/javase/6/docs/api/java/util/HashMap.html

[20] java.apache.commons.lang.builder.HashCodeBuilder:http://commons.apache.org/lang/api-release/org/apache/commons/lang/builder/ HashCodeBuilder.html

[21] Thomas Wang, Jan 1997, Integer Hash Function: http://www.concentric.net/~Ttwang/ tech/inthash.htm