Universiteit
Leiden
The Netherlands

A library for BDDs and SDDs

Michael van der Zwart

Supervisors:
Alfons Laarman
Walter Kosters

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

November 22, 2021

Abstract

Nowadays, decision diagrams are known as important data structures. They are involved in complicated real-world techniques, like model checking and software verification. The Binary Decision Diagram (BDD) has been used for a long time now. However, recently a new type of decision diagram has made an entrance, known as the Sentential Decision Diagram (SDD). Since both decision diagrams complement each other, users need a way for facile exchange of both. To solve this problem, we developed a wrapper library that allows the exchange between the two. Thus, the main objective of this thesis, is exchange of both decision diagrams with ease of play, to allow for facile use of BDDs and SDDs.

Contents

1 Introduction

Boolean functions can be represented using a data structure, named decision diagram, which can be useful for, e.g., the domain of model checking [CG18]. This thesis will deal with two different decision diagrams, known as Binary Decision Diagrams (BDDs) and Sentential Decision Diagrams (SDDs). In some situations there can exist a certain preference for use of BDDs, and in other situations for use of SDDs. This thesis aims to find a way to make BDDs and SDDs exchangeable. Hereby, we make use of Object Oriented Programming (OOP), to facilitate ease use of different types of decision diagrams.

Motivation

There are multiple situations in which both BDDs and SDDs can be used. However, there can exist situations in which it is more effective to use BDDs over SDDs, or vice versa. For example, SDDs tend to be more compact than BDDs [Bov16]. On the other hand, the number of supported BDD / SDD operations in polytime can have an effect on making the choice for one specific decision diagram [DM11]. Thus, selection between both types of decision diagram should be made easy, so that the switch can be made quickly when necessary.

Research Question

Thus, the main question in the thesis, is how the exchange of BDDs and SDDs can be made easy, so that one specific decision diagram can be used, depending on the situation. When use of the other decision diagram is required at a certain time, the switch can be made with ease of play. At the moment, one can make use of BDDs with, for e.g., the Sylvan library, that was provided by Tom van Dijk and others [vD20]. Thus, it could be useful to provide a Sylvan interface for SDDs as well, to add the functionality to also choose for SDDs instead of only BDDs. As a result, the research question for this thesis can be formulated as follows:

> How can BDDs and SDDs be made exchangeable using OOP?

To find a solution to the problem, first a general interface should be defined, which should be generic for both BDDs and SDDs. Both decision diagrams will have the same function-set, but implemented differently. The currently existing BDD interface can be used as template, since the functions therein need to be created for the SDDs too. Thereafter, the functions for the SDD interface need to be implemented. Finally, the interface should be tested. This can be done by, e.g., testing the functions on constraint satisfaction and model checking problems, to evaluate whether they do their expected work, and to validate the interchangeability. A benchmark that we use here, is that the code using our library should be independent of the choice of the decision diagram.

Approach

To create the interchange object, an abstract interface is developed. For each decision diagram type, this interface needs to be implemented, so it represents a template for decision diagram classes. As a result, BDDs and SDDs make use of the same function-set, but implement the functions differently. Thus, each function from the interface is implemented for BDDs in the `Bdd` class, and for SDDs in the `Sdd` class. Both decision diagram types also need a class for representing a set of decision diagram variables, with which variable sets can be constructed. For this, we also develop an abstract interface `VariableSet`, with functions that are implemented differently for BDDs and SDDs. The `Bdd` class has friend class `BddSet`, and the `Sdd` class has friend class `SddSet`. Classes `Bdd` and `Sdd` are conditionally wrapped into a general type `dd`, which can be used for both decision diagrams with only switching a Boolean constant. We also need a wrapper type for `VariableSet` named `vset`, and a wrapper type for decision diagram variables named `ddvar`.

Contribution

This thesis delivers a decision diagram library, in which BDDs and SDDs are interchangeable with ease of play. The same code can be run for BDDs and SDDs, and a simple switch of a Boolean variable results in selection between both decision diagram types.

Overview

This is a thesis submitted in fulfillment of the requirements for the degree of BSc, in the Leiden Institute of Advanced Computer Science. The thesis was supervised by Alfons Laarman and Walter Kosters. The structure of this thesis:

In Chapter 2, BDDs and SDDs are explained. A detailed description of the currently existing BDD interface is also provided, and how to use it. Chapter 3 is about the related work of the topic, which gives a better insight into the problem. The main contribution of this thesis is presented in Chapter 4, to make clear what the value is of the thesis, and what will be delivered. It also describes the structure of the interchange object. Chapter 5 describes the the API of the object, and how to work with it. In Chapter 6, the experiments with the object are shown, and the results are evaluated. Finally, Chapter 7 concludes the thesis, and states further research that could be done.

2 Background

In this thesis, the decision diagram is central. The decision diagram is a data structure to represent a Boolean function. We make use of two types of decision diagrams, known as Binary Decision Diagrams and Sentential Decision Diagrams. This chapter discusses both.

2.1 Binary Decision Diagrams

Structurally, Binary Decision Diagrams (BDDs) are directed acyclic graphs (DAGs). The graph exists of vertices that are labeled with Boolean variables. Each vertex has two edges for heading over to a next vertex, known as a true and a false edge, and each edge means assignment of the Boolean variable represented by the vertex. When heading over the false edge, it means the variable is false. When heading over the true edge, the variable is true. As a result, following a path from root to leaf, will result in an assignment of the Boolean variables in the BDD.

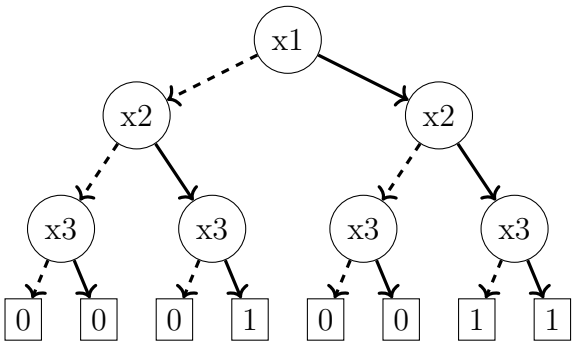

Figure 1: BDD representing $(x_1 \wedge x_2) \vee (x_2 \wedge x_3)$

In Figure 1, an example of a BDD representing a Boolean function is shown. The BDD represents true precisely when x_1 and x_2 are true or when x_2 and x_3 are true. Following the bold line means the variable is true, and following the dashed line means the variable is false. An example of a satisfying assignment is shown in Figure 2.

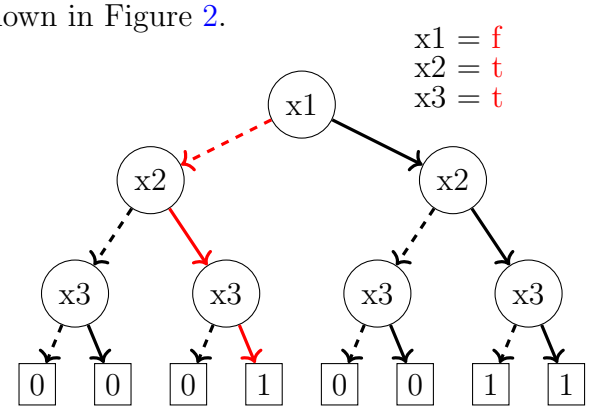

Figure 2: Satisfying assignment of BDD representing $(x_1 \wedge x_2) \vee (x_2 \wedge x_3)$

Now for, e.g., when x_1 and x_2 are true, it does not matter anymore what x_3 evaluates to, since the function will always represent true. To deal with this overhead, the Ordered Binary Decision Diagrams (OBDDs) is used, which will be discussed in next section.

2.1.1 Ordered Binary Decision Diagrams

As discussed in Section 2.1, variables can occur in a BDD that have no effect anymore on the evaluation of the BDD. These variables are called overhead, and are not needed in the BDD. To make the structure canonical, the BDD needs to be ordered, otherwise it will lead to more memory usage, which is unnecessary. The result is an Ordered Binary Decision Diagram (OBDD). Each OBDD is unique for a given variable order, if the OBDD is reduced. When x_1 and x_2 are true, then the BDD will always evaluate true. On the other hand, when x_1 and x_2 are false, then the BDD will always evaluate false. Thus, the BDD can be ordered, as shown in Figure 3, by swapping x_1 and x_2, and removing unnecessary parts. Here, variable ordering [x2, x1, x3] is used. It represents the same Boolean function, but uses less memory.

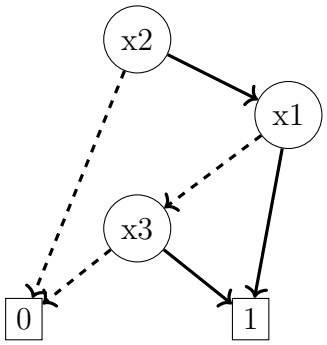

Figure 3: OBDD representing $(x_1 \wedge x_2) \vee (x_2 \wedge x_3)$

2.2 Sentential Decision Diagrams

The Sentential Decision Diagram (SDD) is, just like the BDD, a representation of a Boolean function. However, SDDs make use of a variable tree instead of variable ordering, which results in more reduction. This is exactly what makes SDDs interesting to use, due to minimal memory usage. On the other hand, some operations can be more costly, which is rather explained in Chapter 3. The currently existing library for constructing and manipulating SDDs in `C` [CD18] is open-source, which allows us to make use of it. The structure of an SDD can be illustrated as in Figure 4.

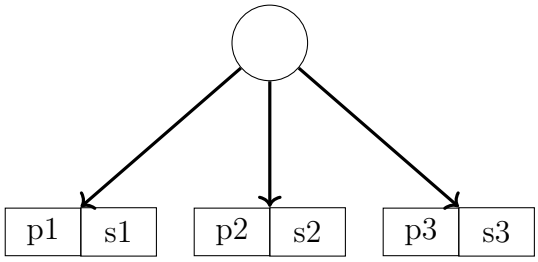

Figure 4: Structure of an SDD

The main difference is, that in BDDs there always exist exact two decisions at each node, say X and $\neg X$. However, in SDDs, branching is done based on sentences instead of variables, called primes (p). Depending on which prime is true, we take the corresponding sub (s).

Circle nodes can be thought of as a Boolean disjunction, and rectangles with primes and subs as Boolean conjunction. Thus, the satisfying assignment of the SDD example can be formulated as

$$(p1 \wedge s1) \vee (p2 \wedge s2) \vee (p3 \wedge s3)$$

Hereby, primes are mutually exclusive and exhaustive, which means that only one prime can be true at the same time. Then, the prime conjoined with the corresponding sub represents a satisfying assignment of the SDD. Thus, Boolean disjunctions in the SDD are exclusive disjunctions.

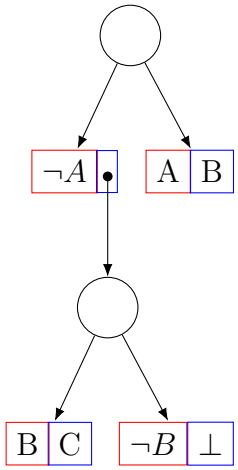

Figure 5: SDD representing $(A \wedge B) \vee (B \wedge C)$

In Figure 5, an example of a SDD is shown. It represents the same Boolean function as the BDD in Figure 1, but has a total different structure. Primes are colored red, and subs are colored blue.

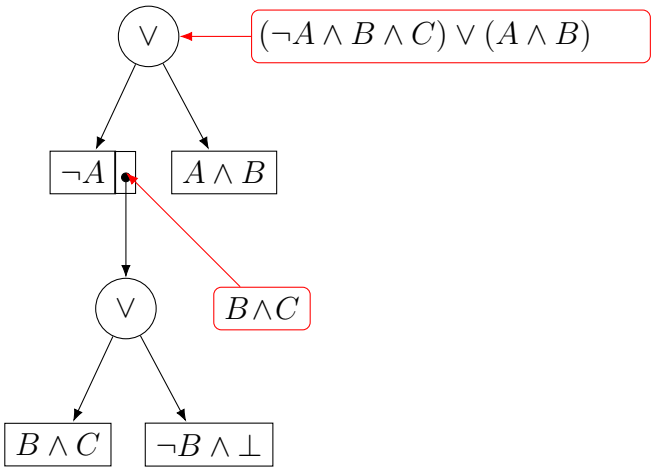

Figure 6: Elaborated SDD representing $(A \wedge B) \vee (B \wedge C)$

Figure 6 illustrates the same SDD as Figure 5, but further elaborated, to clarify the structure of it. We found satisfiable assignment

$$(\neg A \wedge B \wedge C) \vee (A \wedge B)$$

Which can be simplified to

$$(A \wedge B) \vee (B \wedge C)$$

2.3 A typical DD library interface

An example of an BDD library is the Sylvan interface [vD20], which has several functions for constructing and manipulating BDDs. Sylvan implements typical binary decision diagram operations also found in libraries like CUDD [Som16], but provides scalable parallel execution of these operations and is more versatile thanks to supporting custom decision diagram terminal types. The BDD-header consists of two classes. The first one is the `Bdd`-class. In this class, BDDs can be constructed and manipulated, and the resulting BDD is maintained as a member variable which is constantly updated. The second class is the `BddSet`-class, which is used to maintain a set of variables. In this class, a couple of functions are defined for performing operations on the set of variables. In short, this section highlights different parts of the BDD library.

2.3.1 Initialization of BDDs

Functions for initialization of BDDs can be used to create terminal nodes, which return a new `Bdd` object. Two different types are defined:

- `bddOne()`: obtain a BDD representing true, one set of satisfiable assignments
- `bddZero()`: obtain a BDD representing false, multiple sets of satisfiable assignments

2.3.2 Manipulation of BDDs

The BDD library defines several operators, which can be used to manipulate BDDs with Boolean operations. Some examples:

- &`-operator`: applies Boolean conjunction on two BDDs. We can use it for constructing satisfying assignments in an BDD
- |`-operator`: applies Boolean disjunction on two BDDs. We can use it for adding satisfying assignments to an BDD
- `!-operator`: applies Boolean negation on an BDD. we can use it for taking the complement of an BDD.

2.3.3 Transition relations and image operations in BDDs

The interface to a BDD-library also supports a function to take the previous or next state of a BDD, based on a relation. Then the user-defined relation is applied to obtain another state of the BDD. In the BDD library, the next state function takes two arguments, namely the BDD representing the relation, and the set of variables. In the BDD library, an even variable in the set represents a source variable (unprimed), and an odd variable a target variable (primed). The source variables represent the variables in the current BDD, and the target variables will be the new source variables when the transition relation is applied. In the SDD library however, variable numbering starts at one, where the BDD library starts indexing variables at zero.

As an example of the transition relation, we can look at moves in the peg-solitaire board game [Peg21]. Peg-solitaire is a one-person board game with, in the traditional setting, 32 pegs and one empty square in the middle. Moves can be done by placing one peg over another in an empty square, and remove the middle peg. The game ends when there is no move left (lose), or when there is only one peg left (win). Board games are a decent example of experimenting with the interface, since verification tools and for, e.g., quantum systems have equal calculation steps. In Figure 7, the initial board is shown. We can represent a peg-solitaire board as a Boolean function with BDDs, by defining a true variable for a peg, and a false variable for an empty square. The trans ition relation can be depicted as shown in Figure 8.

Figure 7: Initial peg-solitaire board; image from [Peg21]

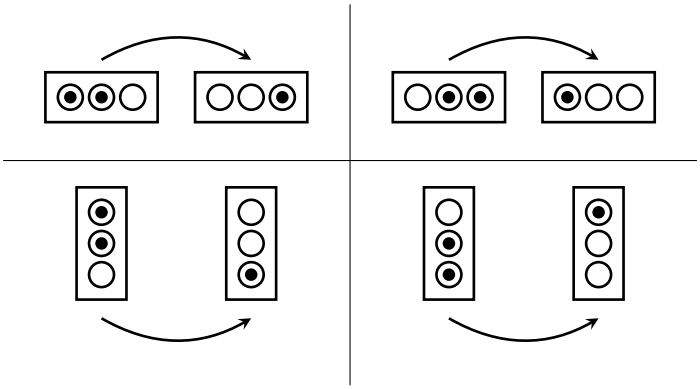

Figure 8: Peg-solitaire move relation

Each triple of horizontal or vertical consecutive squares on the board can be involved in a possible move. Thus, the transition relation can be defined as conjoining sources x_i, x_j and x_k, with i, j, k the index of the variable in the triple of source variables, with their target variables x'_{i+1}, x'_{j+1} and x'_{k+1}, where x is unprimed and x' is primed. Note that each source variable i has corresponding target variable $i+1$. For each triple of squares, two moves could be possible, depending on the sequence of pegs and empty squares in the triplet. In code, this needs to be defined as

$$\underbrace{\underbrace{(x_i \wedge x_j \wedge \neg x_k)}_{SourceVars} \wedge \underbrace{(\neg x'_{i+1} \wedge \neg x'_{j+1} \wedge x'_{k+1})}_{TargetVars}}_{move1}$$

$$\underbrace{\underbrace{(\neg x_i \wedge x_j \wedge x_k)}_{SourceVars} \wedge \underbrace{(x'_{i+1} \wedge \neg x'_{j+1} \wedge \neg x'_{k+1})}_{TargetVars}}_{move2}$$

conjoining sources with targets to form a move of the relation. For each move, all remaining source variables still need to be added to the relation, since they were not involved in the move.

After setting up the transition relation, the `RelNext` function can be used, which conjoins the current BDD with the BDD of the relation and performs existential quantification, to find the satisfiable assignments. This is symbolic model checking [McM92], solving the problem of exponentially growing finite state models as the number of components in the system increases. As function arguments, the transition relation should be provided, and the specific BDD to perform this relation on. Also, the set of relation variables should be provided. The resulting reachable states are returned.

2.4 Reachability with decision diagrams

An important real-world application of BDDs is model checking [CG18], and the use of BDDs has also extended to the domain of software verification. From the paper of Vardi [Var09], we can conclude that model checking can be reduced to a reachability problem, which means that we are actually dealing with a graph-searching problem, in finding all reachable states. We can symbolically represent the system of states and transitions in a DD. For discovering all reachable states inside the DD, several algorithms exist. Algorithm 1 illustrates finding all states with Breadth First Search (BFS), which is a search algorithm that first expands all nodes of the current level, before heading over the next level. With the BFS algorithm, we can expand decision diagram nodes, until all reachable states are found.

Algorithm 1 Reachability

```
1: procedure BFS(InitialStates)          ▷ Generate all reachable states from initial states
2:     Next ← InitialStates              ▷ Initialize BDD with initial states
3:     Visited ← InitialStates
4:     while Next ≠ ∅ do                 ▷ Until all reachable states found
5:         Visited ← Visited | Next      ▷ Add found states to visited states
6:         Next ← Next.RelNext(Transition, Variables)   ▷ Find next states (if there)
7:     end while
8:     return Visited                    ▷ Return all visited states
9: end procedure
```

Thus, with the algorithm, we can discover all reachable states of the symbolically represented system in a decision diagram. Each iteration, the transition relation is applied, until no more undiscovered states are found.

3 Related Work

In this section, we mention some related work associated with the thesis, and use it to motivate our work.

3.1 The origin of BDDs

BDDs were first introduced by Lee and Akers [Lee59], in 1959. In this paper, the relationship between Binary Decision Diagrams and Switching Circuits is shown. Additionally, it shows that the representation of a Boolean function in a Binary Decision Diagram, is superior to the usual Boolean representation. Additionally, BDDs were further developed by Bryant [Bry86], who also introduced polynomial manipulation operations on BDDs.

3.2 Operation ordering problem

In a paper on threshold BDDs [Beh07], Behle mentions that BDDs can be used for finding all feasible solutions of a constraint satisfaction problem, and the optimal solution to a given linear objective function, in a very effective way. However, he also discloses the main problem on BDDs, known as the memory threshold. Behle states that when the operations are ordered in a wrong way, BDDs tend to explode in size. This is known as a huge drawback, since finding the optimal order is a NP-hard problem.

3.3 SDDs

In 2011, Darwiche [Dar11] invented a new decision diagram type named SDDs, and proved that every OBDD actually is an SDD. This is very promising for the SDD data structure, since it is an already ordered BDD, and thus prevents the NP-hard problem of ordering the operations. Later on, Bova [Bov16] theoretically proved that SDDs are even more succinct than OBDDs, which really encourages selecting SDDs over BDDs when it is advantageous to keep memory usage low.

3.4 Selection between decision diagrams

In a paper from 2011 written by Darwiche [DM11], the succinctness of the used data structure is compared to the amount of transformations it supports in polytime. For example, we can have an SDD which is more succinct than the BDD, but the BDD supports more transformations in polytime, which are not supported by the SDD. As a result, when making a choice between what data structure to use, a balance should be found between succinctness and the number of operations supported in polytime. Darwiche concludes that the designer will have to find the most succinct data structure, that supports the necessary operations in polytime. Thus, it is situation dependent which specific decision diagram to select, which encourages a library that supports use of both, with facile interchange.

4 A library for BDDs and SDDs

This chapter provides a description on how we built the interchange library, and illustrates the structure of it via an UML class diagram. The creation of the Sylvan interface for SDDs is also explained.

4.1 Interface setup

In this section, the structure of the interchange object is discussed, including an UML class diagram.

4.1.1 Abstract class

In order to facilitate a seamless exchange between BDD and SDD implementations, Object Oriented Programming (OOP) is used. Dependent on what type of decision diagram is used, bdd or sdd object are created. To obtain this, a super class `DD` is developed, which is an abstract class. The super class `DD` contains the virtual function declarations, which are implemented differently for the two types of decision diagrams, BDDs and SDDs.

4.1.2 Instances

For the implementations of the virtual functions from the super class, we create two sub classes `Bdd` and `Sdd`, which implement the functions from the super class, but differently. For creation of a specific instance, we make use of wrappers. As we follow the structure of Sylvan BDDs library, we can not create objects of the abstract interface, since the objects are allocated on the stack. As a result, BDD and SDD types are compiled into a general type, which can be switched with a Boolean variable present in `dd_exchange.hpp`. The general type can be use, resulting in interchangeability with just switching the Boolean constant.

4.1.3 UML class diagram

Figure 9 illustrates the structure of the interchange object in the form of an UML class diagram. An abstract interface `DD` is illustrated, which is implemented by classes `Bdd` and `Sdd`. The implementation classes also have parts only available within the specific class. Both implement the operators from the abstract class. They also have their own functions, because they are specific to the decision diagram type. The `Bdd` class needs a function for initialization to set up the Sylvan package. To maintain sets of variables, the abstract super class `VariableSet` is created. Again, methods of the class are implemented by classes `BddSet` and `SddSet`, which both have their own decision diagram attribute, `bdd` and `sdd`.

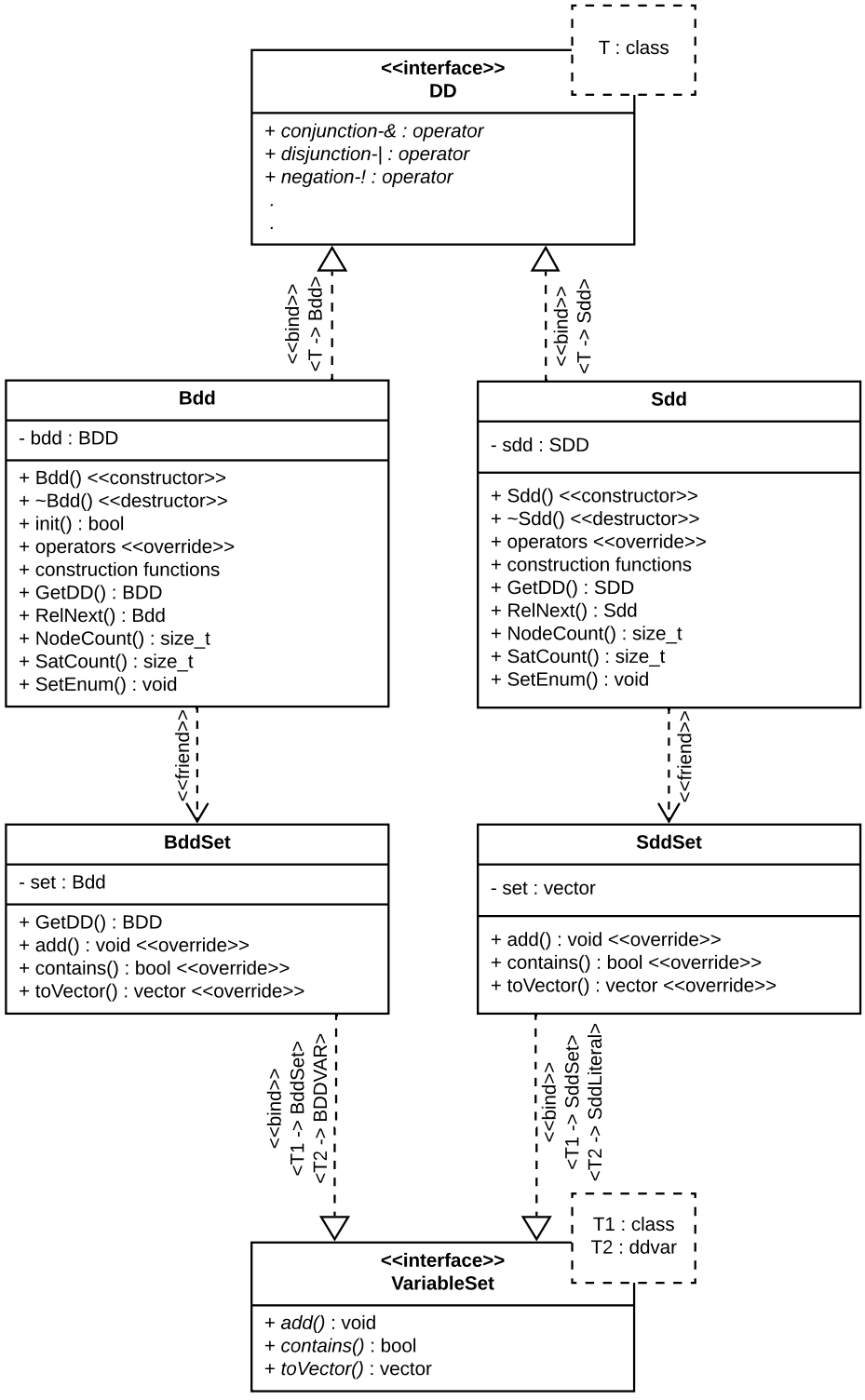

Figure 9: Interface Object: Class Diagram

4.2 SDD interface

To develop the sylvan interface for SDDs, the same functions are used, but implemented differently. In this section, the differences are highlighted. For the implementation of SDD functions, we make use of the SDD package [CD18].

4.2.1 Manager

The SDD package uses a Manager, which should be involved when constructing and manipulating SDDs. The Manager should have twice the size of the number of variables needed, since each source variable also has a target variable. Managers can be created with the `sdd_manager_create` function.

4.2.2 Variable numbering

For BDDs, variable numbering starts at zero. Thus, source variables have an even index, and target variables have an odd index. However, the SDD package starts variable numbering at one, resulting in odd source variables, and even target variables. This should be taken into account when making use of the library, since the same code is run for both BDDs and SDDs. Incorrect numbering of variables will result in aborts for SDDs, since creating a variable with index zero is not allowed.

4.2.3 Transition relation

Since the SDD package does not include a function to compute next state of an SDD, this function should be created from scratch. As function parameters, we need the SDD to perform the next state computation on, and the `SddSet` with the relation variables. The SDD package does include functions needed for existential quantification, known as `sdd_exists_multiple` and `sdd_rename_variables`. At first, the current state of the SDD needs to be conjoined with the SDD of the relation. Then we get the satisfying assignments with the `sdd_exists_multiple` function, which needs an exists_map, which is basically a map with all indices of source variables set to 1. Then we put the result in the `sdd_rename_variables` function, which also needs a map from target variables to source variables. The result contains the next states (if there), and this is returned.

4.2.4 Set enumeration

To print out satisfying assignments of a decision diagram, a function `set_enum` should be developed. The Sylvan package does not include a function to iterate through all possible assignments, so this function should be developed for both BDDs and SDDs. For BDDs we can make use of already existing sylvan functions. However, the SDD package does not include functions for iterating through satisfying assignments. To solve this, we reuse parts of an open source implementation of an SDD-based model checker [Vin18]. The code is adapted a bit, and can be found in the `sdd_sat_enum.h` file on the GitHub [vdZ21]. A callback to a print function can be provided to the `set_enum` function, which prints out the satisfying assignments according to the callback function.

5 Interface API

This section discusses the API of the interchange library, including illustrations of using the API.

5.1 Interchange

In the interface header `dd_exchange.hpp`, a `static int` variable is present which configures use of BDDs or SDDs, which works like demonstrated below.

```
static int dd_choice = 0; //SDD
static int dd_choice = 1; //BDD
```

With the variable set to zero, SDDs are used. Mapping the variable to one will result in use of BDDs. The variable can be changed by the user and then compiled into a working BDD or SDD object.

5.2 Wrappers

The interface header `dd_exchange.hpp` makes use of conditional wrapping of variable types, so that BDDs and SDDs can be interchanged with the tiniest change in code. For this, `C++` conditional typedefs are used.

```
typedef conditional<dd_choice,Bdd,Sdd>::type dd;
typedef conditional<dd_choice,BDDVAR,SddLiteral>::type ddvar;
typedef conditional<dd_choice,BddSet,SddSet>::type vset;
```

Dependent on `dd_choice`, BDD or SDD types are compiled into a general type. When using the general type in code, interchange between BDDs and SDDs can be done by only switching the `dd_choice` variable.

5.3 Example program

In the code below, an example is illustrated of working with the interface. This code can be run for both BDDs and SDDs by simply switching the variable `dd_choice`.

```
dd one = dd::ddOne();
dd zero = dd::ddZero();

dd a = dd::ddVar(1);
dd b = dd::ddVar(2);

one = a&b;
zero = a|b;
```

6 Evaluation

The source code written for the exchange object can be found in the GitHub repository [vdZ21]. The project that can be found there, is a stand-alone project. When installed, the libraries will automatically be found by the project. The GitHub also includes a README file, in which all information about the setup can be found. In this chapter, the exchange object is validated with examples, that test the library.

6.1 Experimental Setup

For the evaluation, we make use of the `C++ assert.h` header. With this header included, statements can be checked on correctness. For, e.g., the statement

assert(X == Y)

will not give any output when X equals Y. However, when X is unequal to Y, the statement will result in *Assertion 'X == Y' failed.*

6.2 Validation

This section provides several example code pieces that test the interchange library. The complete test program can be found in the GitHub.

6.2.1 Initialization of decision diagrams

The interface provides function to obtain terminal decision diagrams, which can be true or false. With the example program below, we can test whether a true DD actually is a true DD, and the same for a false DD. We can also test if they are unequal, and if they complement each other.

```
dd one = dd::ddOne();
dd zero = dd::ddZero();

assert(one != zero);
assert(one == !zero);
assert(!one == zero);
assert(!(!one) == one);
assert(!(!zero) == zero);
```

6.2.2 Manipulation of decision diagrams

For testing the different operators of our library, we can make of Boolean Algebra. For, e.g., distributivity rules can be applied, as demonstrated in the code piece below.

```
dd a = dd::ddVar(1);
dd b = dd::ddVar(2);
dd c = dd::ddVar(3);

assert((a&(b|c)) == (a&b|a&c));
assert((a|(b&c)) == (a|b&a|c));
```

6.2.3 Transition relation

For validating the transition relation, we can define two different decision diagram states X_1 and X_2, and define relation R such that $(X_1, R) \rightarrow (X_2)$. Then we can perform a statement check on whether (X_1, R) is actually equal to X_2. For, e.g. we can have a previous state [0,0] and a next state [1,1] defined as

```
dd state_prev = dd::ddOne();
state_prev = !dd::ddVar(vars[0])&
             !dd::ddVar(vars[1]);

dd state_next = dd::ddOne();
state_next = dd::ddVar(vars[0])&
             dd::ddVar(vars[1]);
```

With the `vars` list containing source variables. Thus, `state_prev` contains two false variables, and `state_next` contains two true variables. Then the transition relation should map state [0,0] to state [1,1] which works like

```
dd relation = !dd::ddVar(vars[0]) &
               dd::ddVar(vars[0]+1) &
              !dd::ddVar(vars[1]) &
               dd::ddVar(vars[1]+1);
```

Which should map false variables to true variables. Finally, we can validate whether the transition relation actually works, by using the `RelNext` function.

```
assert(state_prev.RelNext(relation, variables) == state_next);
```

6.2.4 Reachability

Assuming the transition function works correct, we can transit through all reachable states. In the code demonstration below, all reachable states are visited using BFS, starting at a certain `FixPoint`, which contains states. For the peg-solitaire example, all boards are visited. As a result, the game can be solved.

```
dd next = FixPoint;
dd visited = dd::ddZero();
while (next != dd::ddZero())
{
  visited |= next;
  next = next.RelNext(MoveRelation, RelationVars);
}
return visited;
```

6.3 Discussion

Executing the test file from the GitHub, which performs tests on the interchange library, can be done without complications. Thus, all statement checks are succeeded, which indicates the correctness of our library.

7 Conclusions and Future Work

The main objective of the thesis was to develop a C++ library which allows for exchange of BDDs and SDDs. We can conclude that we have created a library [vdZ21] where switching between BDDs and SDDs can be done with ease of play. The user is able to exchange the two types of decision diagram with ease of play, and can pick a best-fit decision diagram when required in a specific kind of situation. The experiments from the example folder have shown that the implementation is actually working, since no error messages are shown when running them. In the examples folder, the peg-solitaire example is also present. All reachable states can be found, and thus the game can be solved for both BDDs and SDDs.

Regarding subsequent projects, to further experiment the interface with regard to real-world problems, we could think of for example software verification, or model checking. These are techniques related with decision diagrams, so the interface could be used for experimenting to test whether it also works to support these kind of problems, in the sense of exchanging the decision diagrams when required.

References

[Beh07] Markus Behle. On threshold BDDs and the optimal variable ordering problem. In *Combinatorial Optimization and Applications*, pages 124–135, 2007.

[Bov16] Simone Bova. SDDs are exponentially more succinct than OBDDs. *CoRR*, abs/1601.00501, 2016.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[CD18] Arthur Choi and Adnan Darwiche. The SDD package. http://reasoning.cs.ucla.edu/sdd/, (Date accessed: 10.05.2021), 2018.

[CG18] Sagar Chaki and Arie Gurfinkel. BDD-based symbolic model checking. In *Handbook of Model Checking*, pages 219–245. 2018.

[Dar11] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, page 819–826, 2011.

[DM11] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *CoRR*, abs/1106.1819, 2011.

[Lee59] C. Y. Lee. Representation of switching circuits by binary-decision programs. *The Bell System Technical Journal*, 38(4):985–999, 1959.

[McM92] Kenneth L. McMillan. Symbolic model checking: an approach to the state explosion problem. https://apps.dtic.mil/sti/pdfs/ADA250924.pdf, 1992.

[Peg21] Peg Solitaire — Wikipedia, The Free Encyclopedia. "https://en.wikipedia.org/wiki/Peg_solitaire", (Date accessed: 30.08.2021), 2021.

[Som16] Fabio Somenzi. CUDD package. https://github.com/ivmai/cudd, (Date accessed: 10.11.2021), 2016.

[Var09] Moshe Y. Vardi. Model checking as a reachability problem. In *Reachability Problems*, pages 35–35, 2009.

[vD20] Tom van Dijk. Sylvan, multi-core Decision Diagram library. https://github.com/utwente-fmt/sylvan, (Date accessed: 25.10.2021), 2020.

[vdZ21] Michael van der Zwart. Bachelor project. https://github.com/mrvanderzwart/DDinterchange, November 2021.

[Vin18] Lieuwe Vinkhuijzen. LTSmin library. https://www.zotero.org, (Date accessed: 25.08.2021), 2018.