

Assignment for VDS Class Project

Lucas Deutschmann, Philipp Schmitz

1 Task Overview

In the course of this Lab assignment you are going to implement a minimal BDD package in C++. This package shall implement the fundamental manipulation methods for ROBDDs as they were introduced in the lecture *Verification of Digital Systems* by Prof. Kunz. The package will be implemented using the *Test Driven Development (TDD)* paradigm. In the assessment we are going to check, whether your code is properly covered by tests and furthermore use our testing library to check for errors.

The project is split into three parts:

1.1 Part 1

In the first part, your task is to implement the basic functionality of the BDD package using the TDD methodology. This is going to be the biggest part of the project, as you are also required to maintain certain workflows, best practices and code documentation styles. To be more specific, you will learn how to

- Set up and maintain a Git repository
- Use CMake as a build system of your project
- Verify your code using GTest
- Set up a Continuous Integration (CI) pipeline
- Create and maintain a documentation for your API.

Part 1 has to be presented by **12.12.2025**.

1.2 Part 2

In the second part, you are going to improve the performance of your implementation. You will learn how to identify performance bottlenecks within your code and how to overcome them. A set of benchmarks will be provided by us, which you can use to track your performance. During this process, your code is constantly checked by the test suite you developed in part 1. Once your implementation is faster than our given threshold and stays within a reasonable memory bound, you pass the second part. Further information on this part will be provided after part 1. Part 2 has to be presented by **16.01.2026**.

1.3 Part 3

In the last part of the Class Project, you will be tasked to extend your implementation further. The goal is to implement an algorithm to check, if for a given state machine and initial state, a certain state is reachable or not. For this part, the tests will be provided by us. Further information on this part will be given to you after part 2. Part 3 has to be presented by **06.02.2026**.

2 Getting started

2.1 Operating System

We *strongly recommend* that you use a Linux distribution (e.g. Ubuntu) for your project, as it makes setting up the project much simpler. If you want to use Windows, you should use the *Windows Subsystem for Linux (WSL)*.

2.2 Git

Your first step should be to set up your group's gitlab repository. Git is a very popular version control system which most of you probably already have experience in. If you have never worked with git before, there is plenty of very good tutorials that can help you get started. The name of your gitlab repository is *vdscpXX*, where XX stands for your group number.

Start by following the instructions in the e-mail to set up the ssh key and clone your repository.

Fig. 1 shows the basic file structure of the project. In the *Readme.md* you should describe what your project does, how to build and use it and how to run the tests. You should use *doc* to manage the documentation and assignments of the different parts. All the source files should be found inside *src*. Build files as well as IDE files must not be added to your repository! You can also add a *.gitignore* file to make sure you don't accidentally add unwanted files.

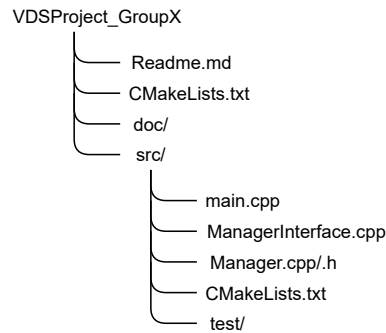


Figure 1: File Structure of the Project

2.3 IDE

You are free to use any IDE you are comfortable working with. We suggest using *CLion* by JetBrains, as it provides a great variety of helpful features. Furthermore, tools required for this project (e.g. git, cmake) can be easily integrated into the IDE. They offer a free student license if you sign up with your RHRK-Account.

2.4 CMake

CMake is a build system, supporting many features like hierarchies, libraries or cross-platform compilation. It is used in many industrial applications and will also manage the build structure for our project. The CLion IDE automatically integrates CMake in every project. In order to compile the project without CLion, cd to build/ and run `cmake ../` which will invoke cmake to generate the makefiles for your project. Running `make` invokes gcc to compile your project. The initial CMake configuration allows you to compile the initial configuration. After adding files to project you need to add them to CMake accordingly. You can find an introduction to CMake here: <https://cmake.org/cmake/help/latest/guide/tutorial/index.html>

2.5 Team Organization

Separate the responsibilities within your team and assign clear roles to everyone. In a two-person team, one person should be responsible for writing tests while another person implements the functionality. If you are in a group of three, think of a way to divide the tasks evenly among your team. Ideally, the person testing the code never does any implementation of the functionality. In the Part 1 sign-off, you will be asked about your team's organization and how you perceived it.

3 Test Driven Development (TDD)

You **are required to** use a TDD flow when implementing your code. Whenever you implement a new feature:

1. Write a unit test for the desired functionality
2. Find the simplest implementation that lets this test pass
3. Extend your test, in case you missed functionality

3.1 Google Test

Tests are implemented using *GoogleTest (GTest)*. Google provides a good documentation that may be used as a starting point. Source code and documentation can be found here: <https://github.com/google/googletest> GTest is integrated into the project via CMake.

3.2 Continuous Integration (CI)

Implementing your code using a TDD flow is a **mandatory** part of the project. Hence, your workflow must be visible for the supervisors. For each new test, there should be a commit before implementing the functionality (test fails) and a commit after implementing the functionality (test passes). During the review meetings, you must be able to explain to your supervisors, what you achieved with the individual commit.

The easiest way to implement your TDD flow is to use *Continuous Integration*. Every time a new commit is pushed to the server, the pipeline builds your implementation and runs your tests. To finish the first part, you **are required to** have a CI Pipeline in place.

4 Documentation

In general, you should use variable, attribute and method names that arouse correct expectations about their use and behavior. In case you feel that a part of your code is difficult to understand, try to make it more readable by using comments. If you put comments inside of your code, they should explain *why* your implementation looks like this, rather than what you did. However, do not annotate each and every line of your code! Most of your code should be understandable by itself. Finding the right balance of commenting is something that comes with experience.

```
/**
 * Sum numbers in a vector.
 *
 * @param values Container whose values are summed.
 * @return sum of `values`, or 0.0 if `values` is empty.
 */
double sum(std::vector<double> & const values) {
    ...
}
```

Figure 2: Example of a function's documentation block

Every non-trivial function you implement should come with a short summary of its functionality, parameters and return values. Fig. 2 shows an example. Feel free to use *Doxygen* or some other tool of your choice to generate docs for your API (optional). For more information, you can refer to: <https://developer.lsst.io/cpp/api-docs.html>

5 Specification

5.1 Data Structure

Your first task will be to implement all functions of the given interface, i.e., *ManagerInterface.h*. The main class of your project will be called *Manager* and inherits the functions from the interface. You are free to create extra classes or functions you need for your project.

	BDD_ID	High	Low	TopVar
False	0	0	0	0
True	1	1	1	1
a	2	1	0	2
b	3	1	0	3
a+b	4	1	3	2

Table 1: BDD_ID Example

As presented in the lecture, you will need to save all nodes with their respective high, low and top variable in a table. How you implement this table is, again, up to you. For each entry, a *BDD_ID* represents the ID of a node. Tab. 1 gives you an example of how a table could look like. *False* and *True* are always represented by 0 and 1 and should be added during the construction of the Manager. Note that for the **Variable Ordering**, the variable that is added first has the highest priority!

5.2 Interface Functions

In the following we are going to describe what every method of the interface is doing.

BDD_ID createVar(const std::string &label)

Creates a new variable with the given label and returns its ID.

const BDD_ID &True()

Returns the ID of the True node.

const BDD_ID &False()

Returns the ID of the False node.

bool isConstant(const BDD_ID f)

Returns true, if the given ID represents a leaf node.

bool isVariable(const BDD_ID x)

Returns true, if the given ID represents a variable.

BDD_ID topVar(const BDD_ID f)

Returns the top variable ID of the given node.

BDD_ID ite(const BDD_ID i, const BDD_ID t, const BDD_ID e)

Implements the if-then-else algorithm, which most of the following functions are based on. Returns the existing or new node that represents the given expression. Please refer to the lecture slides for a detailed description.

BDD_ID coFactorTrue(const BDD_ID f, BDD_ID x)

Returns the positive co-factor of the function represented by ID f w.r.t. variable x. The second parameter is optional. If x is not specified, the co-factor is determined w.r.t. the top variable of f.

Example: $f = a + (b * c)$ with alphabetical variable order

$coFactorTrue(f) = 1 = coFactorTrue(f, a)$

$coFactorTrue(f, c) = a + b$

BDD_ID coFactorFalse(const BDD_ID f, BDD_ID x)

Returns the negative co-factor of the function represented by ID f w.r.t. variable x. The second parameter is optional. If x is not specified, the co-factor is determined w.r.t. the top variable of f.

Example: $f = a + (b * c)$ with alphabetical variable order

$coFactorFalse(f) = b * c = coFactorFalse(f, a)$

$coFactorFalse(f, c) = a$

BDD_ID neg(const BDD_ID a)

Returns the ID representing the negation of the given function.

BDD_ID and2(const BDD_ID a, const BDD_ID b)

Returns the ID representing the resulting function of $a * b$

or2, xor2, nand2, nor2 and xnor2 are defined analogously to and2.

std::string getTopVarName(const BDD_ID &root)

Returns the label of the top variable of *root*.

void findNodes(const BDD_ID &root, std::set<BDD_ID>&nodes_of_root)

This function takes a node *root* and an empty set *nodes_of_root*. It returns the set of all nodes which are reachable from root including itself.

void findVars(const BDD_ID &root, std::set<BDD_ID>&vars_of_root)

This function takes a node *root* and an empty set *vars_of_root*. It returns the set of all variables which are reachable from root including itself if root is a variable. *Hint*: It essentially returns the set of top variables of findNodes excluding the terminal nodes.

size_t uniqueTableSize()

Returns the number of nodes currently existing in the unique table of the Manager class.

void vizualizeBDD(std::string filepath, const BDD_ID &root)

Creates a file that contains a visual representation of the BDD represented by the root node. You can be creative with your representation!

This function is excluded from the TDD approach.

Hint: The *dot* format is probably the most convenient way but we are looking forward to different approaches. (see <https://graphviz.org/doc/info/lang.html> and <https://dreampuf.github.io/GraphvizOnline>)

6 Useful Tips for Writing Test Cases

The person(s) responsible for writing the test cases should consider a few points:

1. Write meaningful tests. Do not test the same thing over and over again. The test cases should feel like the documentation of the code, i.e., someone solely looking at the tests should understand what the code does.
2. Think of the different use cases for the functions defined in the interface and try to cover all of their behavior. Also use the True and False entries for testing the logic functions.

3. Take a look at so called Test Fixtures (TEST_F). You can construct a BDD Manager in a structure and initialize it with a few variables that can then be reused by the Test Fixtures.
4. When testing the logic, do not write test cases that expect concrete numbers and use variables storing the IDs instead. For example, $EXPECT_EQ(manager \rightarrow and2(a_id, false_id), false_id)$ is a better test than $EXPECT_EQ(manager \rightarrow and2(2, 0), 0)$.
5. When you think you are done with the Test Driven Development check the coverage metrics provided by your IDE. In CLion, you can let the tests run with coverage by clicking on the "shield" symbol next to your configuration. (See <https://www.jetbrains.com/help/clion/code-coverage-clion.html#run-coverage>)

7 ROBDD Construction Example

We now want to see an example on how the construction of a Reduced Ordered BDD with the above interface works. Detailed information on the underlying algorithms can also be found in the lecture slides.

For this example, we want to build a ROBDD for

$$f = (a + b) * c * d$$

which is equivalent to

$$f = and2(or2(a, b), and2(c, d))$$

in our representation. To create the ROBDD, we perform the following steps:

1. Instantiate the Manager Class. This should also add 0 and 1 to the table.
2. Create variables a, b, c and d using the `createVar("a")` function. The unique table should now have 6 entries.
3. Now, we call $or2(a, b)$ which should internally call $ite(a, 1, b) = ite(id2, id1, id3)$. As this is not a terminal case, we proceed with the ite algorithm and determine the top variable of the given expressions. In our case, a is the top variable. The high and low successor of $a + b$ are determined by

$$highSuccessor = ite(coFactorTrue(id2, a) = 1, coFactorTrue(id1, a) = 1, coFactorTrue(id3, a) = b) = 1$$

$$lowSuccessor = ite(coFactorFalse(id2, a) = 0, coFactorFalse(id1, a) = 1, coFactorFalse(id3, a) = b) = b$$

Both ite calls are terminal cases and therefore immediately resolved. As no entry with $High = 1$, $Low = b$ and $topVar = a$ exists, a new entry (id6) is added to the unique table.

4. The next step is to call $and2(c, d)$ which itself calls $ite(c, d, 0) = ite(id4, id5, id0)$. Again, no terminal case, hence we determine c as the top variable. The high and low successor of $c * d$ are determined, within ite, by

$$highSuccessor = ite(coFactorTrue(id4, c) = 1, coFactorTrue(id5, c) = d, coFactorTrue(id0, c) = 0) = d$$

$$lowSuccessor = ite(coFactorFalse(id4, c) = 0, coFactorFalse(id5, c) = d, coFactorFalse(id0, c) = 0) = 0$$

Both ite calls are terminal cases and therefore immediately resolved. As no entry with $High = d$, $Low = 0$ and $topVar = c$ exists, a new entry (id7) is added to the unique table.

5. The last step is to call $and2(a+b, c*d) = and2(id6, id7)$ which itself calls $ite(id6, id7, id0)$. The top variable of id6 and id7 is a. After the co-factoring, the following ite calls will be made

$$highSuccessor = ite(id1, id7, id0) = id7$$

$$lowSuccessor = ite(id3, id7, id0)$$

While the first call is a terminal case, the second one is not and we enter a recursion:

The top variable of $ite(id3, id7, id0)$ is b (id3) and its successors are

$$highSuccessor' = ite(id1, id7, id0) = id7$$

$$lowSuccessor' = ite(id0, id7, id0) = id0$$

for which we add a new entry (id8) with $High = id7$, $Low = id0$ and $topVar = b = id3$.

The new entry will be the low successor of $ite(id6, id7, id0)$. At last, we create the entry for our function with $High = id7$, $Low = id8$ and $topVar = id2 = a$.

Tab. 2 shows the final unique table for this example. Fig. 3 shows the corresponding ROBDD for the function f. Please note that the labels for the complex nodes are not mandatory.

BDD_ID	Label	High	Low	TopVar
0	<i>False</i>	0	0	0
1	<i>True</i>	1	1	1
2	<i>a</i>	1	0	2
3	<i>b</i>	1	0	3
4	<i>c</i>	1	0	4
5	<i>d</i>	1	0	5
6	$a + b$	1	3	2
7	$c * d$	5	0	4
8	$b * c * d$	7	0	3
9	<i>f</i>	7	8	2

Table 2: Unique Table Example

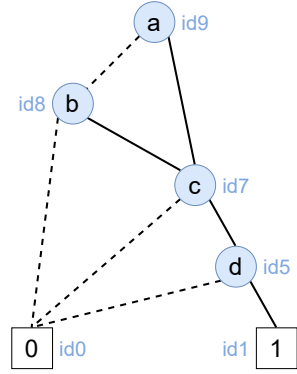


Figure 3: ROBDD Example