Assignment for VDS Class Project

Tobias Ludwig

1 Task

1.1 Part A

Requirements

In the course of this Lab assignment you will 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 System by Prof. Kunz. The package will be implemented using the Test Driven Development pardigm presented by Dr. Wedler. 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.

Documentation
Write a unit test first than find the simplest implementation that lets this test pass. Refactor your implementation as required while you are moving forward. Use variable, attribute and method names that pitch correct expectations about their use and behavior to your users. Write a set of acceptance tests that demonstrate the use of your package. Use doxygen or some other tool of your choice to generate the docs for your API.

For testing we are going to use GTest. Google provided a good documentation that may be used as a starting point. Furthermore, I'd like to point to our fraction example on the website. GTest will be installed on the server, if you want to use your own machine you need to install GTest and link against it.
The source code and documentation can be found on: https://github.com/google/googletest

At the root level of the archive provide a README file with instructions. Explain at least:

- How to build a static library from your source. (e.g.: make Makefile or scons -u or CMake ...)
- How to run the unittests.
- How to run the acceptance tests of the entire package.

As a second artefact you are requested to maintain a **process-log-book**. This document shall describe the development process that you followed and the individual intermediate results that you achieved in the course of this project. The simplest way to fullfill this requirement is to use a revision system (e.g., git) and frequently commit intermediate results with meaning full commit messages.
Important:
You are required! to validate your Test-Driven-Development. For each test, there is a commit with the test before implementing the functionality(test fails) and a commit after implementing the functionality(test passes). For example, if there are 10 tests, there will be 20 commits in git.
Within review meetings with your supervisor, you must be able to explain, what you achieved with the individual submits.

Continuous Integration (CI) with GitHub Actions The easiest way to validate you TDD is to use GitHub Actions. This allows you to set-up an continuous integration pipeline. Everytime a new commit is pushed to the server, the pipeline builds your tool and runs your tests. The first commit on the server should show a failing test. The second the passing test. To finish part A you're required to have a CI Pipeline in place. More information on pipelines: https://lab.github.com/githubtraining/github-actions:-hello-world
Getting started
The project starts with the files from the project website. In the folder there is a file ManagerInterface, which defines an Interface you have to implement. The class that implements the interface has to be named Manager and is also part of the ClassProject namespace. You are totally free on implemention of the project, but the interface MUST stay unchanged. We are going to use the interface to check your code. In Section 2 more information on the interface methods is provided.

The folder structure is the following:

- VDSProject/
    - build/
    - doc/
    - src/
        - main.cpp

ManagerInterface.cpp/.h

Manager.cpp/.h

test/

- build/ is used to run cmake and after running make it will also contain the executables
- doc/ contains the documentation
- src/ contains all source files
- src/test/ This folder contains the tests. Tests can either be added to files Test.cpp, Test.h or self-added files.

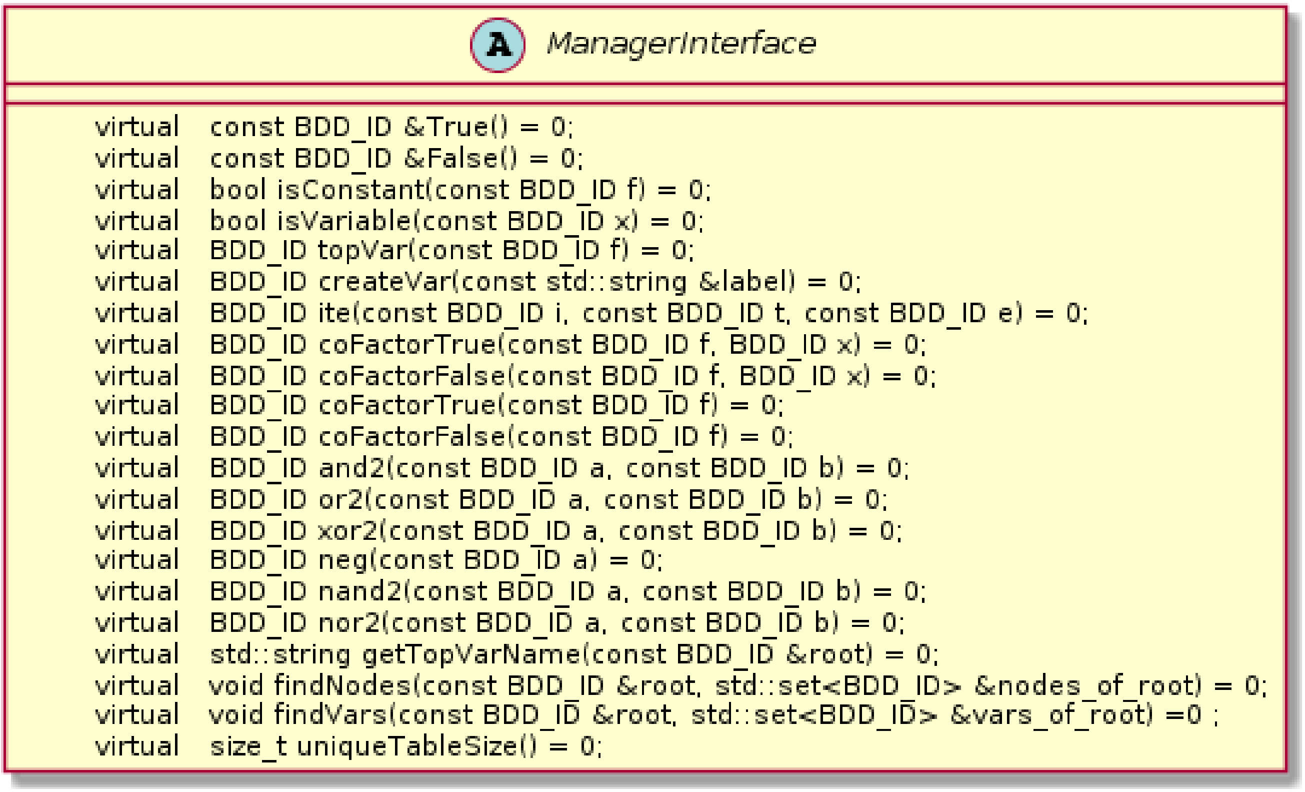
Every folder contains a CMakeLists.txt. CMake is build system build on top of make. In order to compile the project cd to build/ and run cmake ../ which will invoke cmake to generate the makefiles for your project. Running make invokes gcc to compile your project.

The intital CMake configuration allows you to compile the initial configuration. If you add files to project you need to add them to CMake accordingly.

1.2 Part B

After ensuring the correct implementation we are going to work on the performance of your code. We are going to extend your code with our benchmarking tool. The goal is to optimize your code such that it reaches a specific runtime threshold. During this process the code is constantly checked with the test. In the assessment we are going to check, whether your code reaches the threshold. There will be more information available on Part B after Part A is done.

2 Specification



```
virtual   const BDD_ID &True() = 0;
virtual   const BDD_ID &False() = 0;
virtual   bool isConstant(const BDD_ID f) = 0;
virtual   bool isVariable(const BDD_ID x) = 0;
virtual   BDD_ID topVar(const BDD_ID f) = 0;
virtual   BDD_ID createVar(const std::string &label) = 0;
virtual   BDD_ID ite(const BDD_ID i, const BDD_ID t, const BDD_ID e) = 0;
virtual   BDD_ID coFactorTrue(const BDD_ID f, BDD_ID x) = 0;
virtual   BDD_ID coFactorFalse(const BDD_ID f, BDD_ID x) = 0;
virtual   BDD_ID coFactorTrue(const BDD_ID f) = 0;
virtual   BDD_ID coFactorFalse(const BDD_ID f) = 0;
virtual   BDD_ID and2(const BDD_ID a, const BDD_ID b) = 0;
virtual   BDD_ID or2(const BDD_ID a, const BDD_ID b) = 0;
virtual   BDD_ID xor2(const BDD_ID a, const BDD_ID b) = 0;
virtual   BDD_ID neg(const BDD_ID a) = 0;
virtual   BDD_ID nand2(const BDD_ID a, const BDD_ID b) = 0;
virtual   BDD_ID nor2(const BDD_ID a, const BDD_ID b) = 0;
virtual   std::string getTopVarName(const BDD_ID &root) = 0;
virtual   void findNodes(const BDD_ID &root, std::set<BDD_ID> &nodes_of_root) = 0;
virtual   void findVars(const BDD_ID &root, std::set<BDD_ID> &vars_of_root) =0 ;
virtual   size_t uniqueTableSize() = 0;
```

Figure 1: UML of Managerinterface

In the following we are going to describe what every method of interface is doing.
Variable Ordering
The variable that is added first has the highest ranking.

BDD_ID
Integer representing the ID of the BDD Node. Don't change!

True() and False()
Returns the ID of the node representing True and False.
Those are added during construction of the object

isConstant(x)
Returns true if x is a leaf node

isVariable(x)
Returns true if x is a variable

topVar(f))
Returns the ID of top variable of the BDD node f

createVar(label)
Creates a new variable for the BDD

ite(i,t,e)
Implements the if-then-else algorithm.
Returns the new node that represents the ITE.
Here it's really beneficial to start with the tests first!

coFactorFalse(f)
Returns the negative cofactor of the function defined by node f.
Example:
Variable order: a,b,c
$f = a + (b * c)$
$coFactorFalse(f) = b * c$

coFactorFalse(f,x)
Returns the negative cofactor of the function defined by f with respect to function x.
Example:
Variable order: a,b,c
$f = a + (b * c)$
$coFactorFalse(f, c) = a$

coFactorPositiv(f)
Returns the positiv cofactor of the function defined by node f.
Example:
Variable order: a,b,c
$f = a + (b * c)$
$coFactorTrue(f) = 1$

coFactorPositiv(f,x)
Returns the positive cofactor of the function defined by f with respect to function x.
Example:
Variable order: a,b,c
$f = a + (b * c)$
$coFactorTrue(f, c) = a + b$

and2, or2, xor2, neg, nand2, nor2
Returns a BDD node that represent the correlating boolean function
Hint: Those methods are implemented using ITE

getTopVarName(f)
Returns asghar the Name of top variable of the BDD node f

findNodes(root,nodes_of_root)
Returns the set of BDD nodes which are reachable from the BDD node root(including itself).

findVars(root,vars_of_root)
Returns the set of variables which are either top variable of the BDD node root or the reachable nodes from root.
Hint: It essentially returns the set of top variables of findNodes(root).

uniqueTableSize
Returns the number of the nodes currently exist in the unique table of the Manager class