Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Developing a test generation tool for C programs

Master's Thesis

Author
Kristóf Verbőczy

Advisor
dr. Micskei Zoltán

December 14, 2018

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Verbőczy Kristóf*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2018. december 14.

Verbőczy Kristóf
hallgató

Kivonat

Annak ellenére, hogy a C nyelv nem tekinthető újnak, mégis számos alkalmazása van mind a mai napig. Elsősorban beágyazott rendszerekben használják, ahol számít a hatékonyság. Erre a C alacsony szintű memóriakezelése megfelelő. Nem csak, hogy nem lett egy elfeledett nyelv a C, de az IoT (Internet of Things) elterjedésével várhatóan még a mainál is nagyobb körben kerül alkalmazásra.

Láttuk, hogy a C nyelv fő felhasználási területe a beágyazott rendszerek. Ezek nem feltétlenül, de az esetek egy részében biztonságkritikus rendszerekben találhatók. A biztonságkritikus rendszerekre az jellemző, hogy egy meghibásodás katasztrofális következményekkel járhat; emberéletek kerülhetnek veszélybe, vagy súlyos anyagi, illetve környezeti kár keletkezhet. Gondoljunk csak bele, hogy egy repülőgép vezérlő, vagy atomreaktor szabályozó meghibásodása milyen következményekkel járhat.

Az ilyen esetek elkerülésére szolgál a szoftverek verifikációja. Ennek a célja, hogy megtaláljunk hibákat a szoftverben, vagy kimutassuk azoknak a hiányát. Az egyik legnépszerűbb verifikációs módszer a tesztelés. Bár tesztelés esetén nem tudjuk matematikailag bizonyítani, hogy egy rendszer nem tartalmaz hibát, mégis segít megbizonyosodni a szoftver minőségéről. Mivel a tesztelés egy igen elterjedt eljárás, ezért sok különböző módszer alakult ki rá. Ezek közül a diplomamunka a kód-alapú tesztgenerálással foglalkozik.

A kód-alapú tesztgenerálás esetében hozzáférünk a forráskódhoz, teljes mértékben ez alapján történik, automatikusan a tesztesetek generálása. A kód-alapú tesztgenerálás végrehajtására is létezik több különböző módszer, például a szimbolikus végrehajtás, kereső algoritmusok, véletlent használó algoritmusok, annotáció-alapú tesztgenerálás.

Ebben a dolgozatban egy olyan kód-alapú tesztgeneráló eszköz kerül elkészítésre, bemutatásra, amely C nyelven írt forráskódokhoz képes teszteket generálni, ezeket lefuttatni, fedettségi értékeket szolgáltatni. A tesztbemenetek generálására modellellenőrzőt használok, ami instrumentált kódon assert utasításokkal lefedésére van beállítva. Mivel léteznek már hasonló eszközök, azok hiányosságait igyekszem előtérbe helyezni az eszköz készítése során.

Az eszköz elkészítése után egyszerűbb példakódokon, és valós környezetből vett kódokon értékelem az eszközt. Ez alapján az mondható el, hogy az eszköz képes teszteket generálni alapvető C programokhoz, támogatja az egyszerű típusokat, tömböket, struktúrákat, bizonyos szintig a pointereket, de vannak korlátai. A C makróit nem kezeli jól, struktúrákat struktúrákban sem képes kezelni. Tehát a meglévő eszközöket nem sikerült túlszárnyalni.

Abstract

Even though C is not considered a new language, it is still used in several applications up to this day. Primarily, it is used in embedded systems, where performance matters. The low-level memory handling makes C convenient for this. Not only C is not dead language, but also it is expected to be used even more widely, due to the propagation of IoT (Internet of Things).

We have seen, C programs are mostly used in embedded systems. Embedded systems are not necessarily safety-critical systems, but in several cases, they can be found in safety-critical systems. Safety-critical systems are such systems, whose failures may have catastrophic consequences; human lives may be endangered, or severe financial, or environmental damage may arise. Think about the consequences, which occur, when a controller gets out of order in an airplane, or in a nuclear reactor.

Software verification serves for avoiding such cases. It aims at finding the flaws in the software, or proving the absent of flaws. One of the most popular software verification method is testing. Though testing cannot prove mathematically the absent of flaws in a software, it still helps to make sure about the quality of the software. Since testing is a widespread procedure, there are lot of different methods for doing it. This thesis deals with code-based test generation.

In case of code-based test generation, we have access to the source code, automatic generation of the test cases is fully based on the source code. There are several methods to execute code-based test generation, for example symbolic execution, search-based algorithms, random algorithms, annotation-based test generation.

In this thesis a code-based test generation tools is being implemented and presented, which is capable of test generation, execution, and providing coverage results for programs, written in C language. For test input generation model checker is used, which is set to cover assert statements of the instrumented code. Since this kind of tools already exist, I try to pay attention to their incompleteness during the development of the tool.

After the tool is ready, I am going to examine it with both simple example C programs, and real world programs. Based on this, we can say, the tool is able to generate tests to basic C programs. It supports simple types, arrays, structures, pointers at some level, but has limitations. The macros of C, structures inside structures are not handled. All in all, my tool does not exceed existing tools.

Chapter 1

Introduction

Although, nowadays managed and script languages are widespread (e.g. *Java*, *C#* for the prior, *Python*, *JavaScript* for the latter), C programming language is still used by many. [1] [2] C is used mostly in embedded systems. Embedded systems are all around us, they can be found at our homes (fridge, washing-machine, digital clocks, etc.), at the streets (inside cars, buses, traffic lights), at power plants (controllers of different processes). Furthermore embedded systems are getting more and more widespread thank to IoT (Internet of Things), which has much to do with embedded systems, since it is about getting electric devices connected to each other through the Internet. Embedded systems frequently serve in a safety-critical domain, what makes it even more important to examine the quality of the program running on the electric devices. This makes me think, C is going to be with us for a while.

We saw that C is a language worth taking into account. As mentioned earlier the domain, where C is used, is mostly safety-critical systems. Safety-critical means human lives, the environment or huge amount of money is endangered, if the system fails, because of some reason. It is easy to feel the difference between a power plant controller unit and a computer game. I guess the reader would chose the game to crash, if (s)he had to chose one. I am not saying that it is not bad, or annoying if a game crashes, but hopefully nobody gets hurt after that. I hope, I presented the importance of verification of C programs decently.

There are more ways to verify the soundness program, for example using *formal verification*, *static analysis* or *testing*. Each of them has it own advantages and disadvantages, this will be explained later on, at Chapter 2. Although there are more techniques, this thesis is concerned with testing.

Testing has several approaches, e.g. unit, integration, regression, white-box, black-box testing, code-based, model-based test generation. This thesis is concerned with code-based test generation. This is a technique, which uses the source code itself to generate test cases automatically. Through automation it can overcome some problems. For example it saves time for the tester, because the tester does not have to write the test code, which reduces the chance of mistyping something. Moreover it may find test inputs for complex conditions and decisions faster. It may sound like, a good code-based test generation tool can dismiss testers, but this is not case. Although, these kind of tools are able to generate test cases, but a tester will always be needed, who analyses the test cases and test results. This is detailed later on at Chapter 2 and Chapter 7.

[1] `https://insights.stackoverflow.com/survey/2017#technology`
[2] `https://www.tiobe.com/tiobe-index/`

There are already numerous different code-based test generation tools even for C programming language. The tools have different capabilities, this is detailed at Chapter 3.

So it is not original idea to create a code-based test generation tool, not even for C. But there is a possibility to learn from the existing tools and create a tool, which is capable of things the others aren't. This was the plan for this thesis work. If it succeeded will emerge in Chapter 6 and Chapter 7.

C programs aren't always trivial, a lot of things can be done with pointers, C supports multithreading as well. These things are hard to test, so this thesis doesn't take into account every element of C, it only works on a subset of C language.

1.1 Goals

I would like to clarify, what I would like to achieve at the end of this thesis work, because the tool itself is just an artefact. Of course, one of the main goal is to create a code-based test generation tool for C programs, which should not be a lot worse than the existing ones. It would be great if the tool would be better than the previous ones, but maybe it is out of the scope of a master thesis work. So I will be happy, if the tool can support only limited elements of C language, and does not have so many functionality like the existing ones, but I try to implement as much as I can.

Moreover, during the university all of my individual laboratory and thesis work dealt with testing. I want to deepen my knowledge on the area of testing. This involves getting experience with testing of C programs, and also with code-based test generation.

Last, but not least, I want to solve the challenge of designing a complex system from scratch. I have never done that before, in this volume.

1.2 Structure of the thesis

The thesis is structured as follows. Chapter 2 introduces the background to safety-critical systems, testing and code-based test generation. Chapter 3 introduces existing code-based test generation tools, their capabilities, limits. Chapter 4 shows design of the tool, its core structure and the decisions that have to be taken. Chapter 5 gets into the implementation of the tool in details. Chapter 6 evaluates the finished tool by different aspect, and compares it with other test generator tools. Chapter 7 concludes the thesis, tells if my goals were achieved, and shows possibilities for improving the tool.

Chapter 2

Background

Despite the implementation or design seems to be the most interesting part in the creation of a tool, a lot of research has to be done before that, in order to have a wide picture of the chosen topic.

It is important to have knowledge about the domain, in which that the tool is going to be used. In addition, one should look around what kind of techniques and methods are available and which of these fits his/her goal the best. Also it can help, if one is aware of the theoretical limits of a technique, so (s)he can have rational expectations about the abilities of the tool.

In this chapter safety-critical systems are introduced as the application domain of the tool. Furthermore, verification techniques are demonstrated through a motivation example.

2.1 Safety-critical systems

In case of safety-critical systems the failure of the system may have catastrophic consequences. It may endanger human life, cause huge economic loss, sorely damage the environment. List 2.1 shows a far from complete list of safety-critical systems [16].

- **Medical applications:** computer-assisted surgery, dialysis machines, infusion pumps, pacemaker, radiation therapy machines.
- **Everyday life:** fire alarm, fuse, telecommunication.
- **Industrial applications:** financial systems, nuclear reactor control systems, power plants.
- **Transport:** automotive, aviation, railway, spaceflight.

List 2.1: List of safety-critical systems

C language is commonly used in development of critical systems, because it is higher level, than assembly, so it is easier to develop, but it is low level enough to handle resources, such as memory, effectively.

2.1.1 Standards

There are more aims of standardization. Firstly, to simplify the commercial and technological flow, secondly, to reach the agreement of the economic participants. In our case, most importantly, to bring safety and guarantee to the users world-wide by the standardization of a product or a service.

The requirements that have to be met by the system are defined in standards. There are many standards, since the domain of safety-critical systems is wide, and what is fundamental in one domain may not appear in an other. List 2.2 shows some standards from different, but mostly from transportation domain.

There are many organizations world-wide, whose task is to come up with better and better standards, for example ISO (International Standardization Organization) and IEC (International Electrotechnical Commission).

- **IEC 61508** - is an international standard for the functional safety of electrical, electronic, and programmable electronic equipment [11].
- **DO-178B** - Software Considerations in Airborne Systems and Equipment Certification [17].
- **EN 50128** - is currently the only European standard that includes detailed requirements for software for the rail industry and it addresses communication, signalling and control and protection systems [20].
- **ISO 26262** - focuses on the functional safety of electrical and electronic systems in vehicles [18].

List 2.2: List of safety-critical system standards

2.1.2 Embedded systems

Embedded systems are special computer systems. Embedded systems are designed for a well-defined task. In order to function well it has an intensive connection with the outside world. They sense some given parameter of the outside world, and often the intervene with it.

There are some requirements against embedded systems, since they are usually safety-critical, so their malfunction can lead to damage in humans. These requirements are *real time functioning*, *dependability*, *availability*, *safety*, *integrity* and *maintainability* [14].

List 2.1 shows examples for embedded systems. To complete the list, here are some other examples from non-critical domain: *camera, television, printer, (PlayStation or Xbox) controllers.*

2.2 Motivation example

The following sections introduces different verification techniques. This section aims at providing a rather simple example, on which the techniques can be explained in an easier to understand way. Below the specification of the example can be found, followed by its implementation.

Specification The function gets a boundary temperature value and three temperature value, each measured by a different sensor. The function returns a majority verdict about detecting it as dangerous situation. Meaning, if any two sensors say the temperature is higher than the boundary value, then it is a dangerous situation, else it is not.

Implementation The code below is a correct implementation of the specification.

```
1  int isDangerousSituation(int boundaryTemperature,
2              int temperature1, int temperature2, int temperature3) {
3
4    if (temperature1 > boundaryTemperature) {
5      if (temperature2 > boundaryTemperature || temperature3 > boundaryTemperature) {
6        return 1;
7      }
8      else {
9        return 0;
10     }
11   }
12   else if (temperature2 > boundaryTemperature && temperature3 > boundaryTemperature) {
13     return 1;
14   }
15   else {
16     return 0;
17   }
18 }
```

It is notable that even for this simple function, committing an error is easy. For example, in line 12, one can easily write *temperature2*, instead of *temperature3*.

```
else if (temperature2 > boundaryTemperature && temperature2 > boundaryTemperature) {
```

This will be used later, to show the result of techniques for not only correct, but for faulty implementation as well.

2.3 Validation and verification

Validation and verification are processes used to check the correct functioning of a program. There are different aspect for calling the functioning of program correct. This brings the difference between validation and verification. These techniques are similar, Table 2.1 helps, to tell them apart.

Verification	Validation
Am I building the system right?	Am I building the right system?
Objective - can be automated.	Subjective - cannot be automated.
Checks design and implementation faults.	Checks problems in the requirements.

Table 2.1: Validation and verification [15]

Since verification can be automated and checks implementation faults, this thesis is concerned with verification, validation is out of its scope. In the following sections, different verification techniques will be introduced, *static analysis* in Section 2.3.4, *formal verification* in Section 2.3.5 and *testing* in Section 2.3.6.

2.3.1 V-model

The V-model is a life cycle model in software development. It can be viewed as the extension of the waterfall model. It is also characterized by the sequential procedure

execution, meaning each phase has to be finished, before staring the next one. Compared to the waterfall model, in case of the V model the procedures after the implementation are located not downwards, but upwards, shaping the peculiar V shape. This represents well, which testing level belongs to a given development phase. This and the connection of the phases can be seen at Figure 2.3. Usage of V-model is prevalent mostly in the development of safety-critical computer system [1].

Phases [2]

1. **Requirements and specification**
   The first step in this phase is measuring the claims of the users, and creating a document based on it. This document is going to be the base of the final validation. By this the engineers fabricate the specification. Moreover documents are made for the system tests.

2. **Architectural design**
   The architecture of the software is made in this phase. This contains the list of modules, the brief description of the modules, description of the interfaces, dependencies and architecture diagrams.

3. **Detailed design**
   The program specification contains the detailed logics of the modules; tables of database, details of all interface, all dependency, list of error messages, the possible inputs and outputs of a module.

4. **Implementation**
   The source code is made in this phase.

5. **Unit testing**
   Unit test plans are made during the detailed design phase. These test are being executed here, to ascertain that the smallest parts of the system are functioning well.

6. **Integration testing**
   Integration test plans are made during the architectural design phase. These tests prove, that the independently well-functioning units can work together or can communicate with each other.

7. **System testing**
   The whole application is being tested. Both functional and non-functional requirements are being checked. Performance testing, stress testing, regression testing are executed in this phase.

8. **Validation**
   Acceptance tests by the users prove if the application is sufficient for the users' claim and it is ready to be used in real environment.

2.3.2 Coding guidelines

Coding guidelines are sets of rules giving recommendations on way the program is written. It applies to style of coding, for example formatting, naming, and also applies to programming practises, such as constructs, architecture. There are three main categories [15].

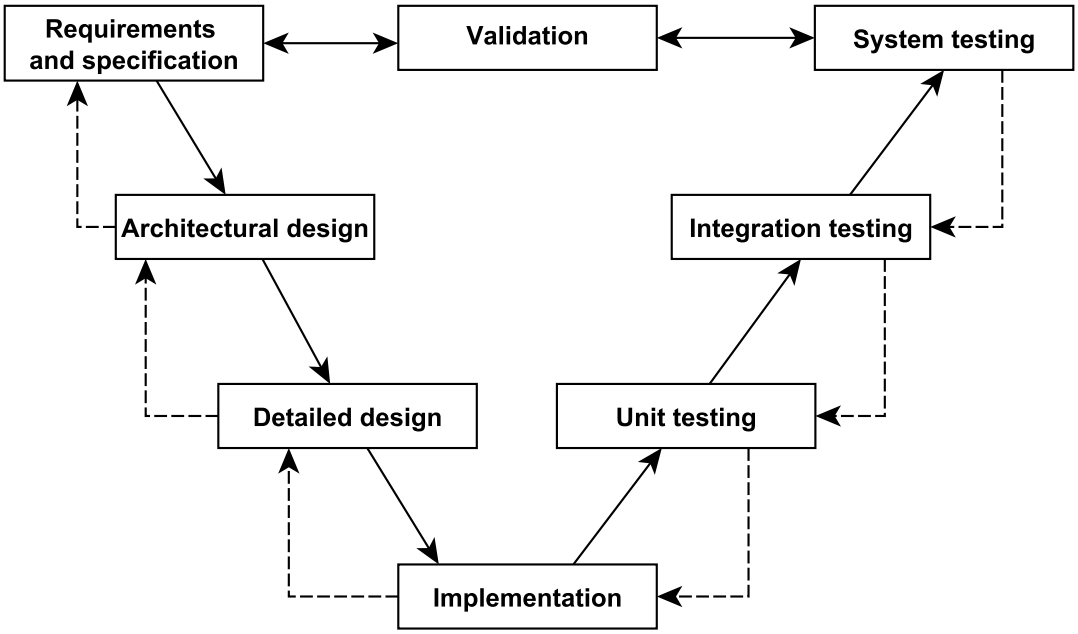


Figure 2.3: V-model [22]

- **Domain specific:** e.g. automotive, aviation, railway.
- **Platform specific:** e.g. Java, C, C++.
- **Organization:** e.g. Microsoft, Google, CERN.

```
array[0] = 109; /* equals to decimal 109 */
array[1] = 052; /* equals to decimal 42 */
```

A concrete example is MISRA C, which is a set of software development guidelines. It is interesting because it deals with C and safety-critical systems. MISRA stands for Motor Industry Software Reliability Association. For example it requires that the code shall only use /* ... */ style comments. It says that the full C language should not be used for programming safety-related systems, for example it forbid the use of octal constants (other than zero) or octal escape sequences. This obviously tries to avoid confusion caused by the mixture of decimal and octal numbers [21].

2.3.3 Code review

Code review is a technique performed by humans, typically other team members, usually based on some structured check list. It may not only reveal bugs, but can improve the understanding of the code, and awareness of a team.

For example GitHub supports code reviews, pull requests can be asked to be reviewed by others, and the message section is convenient for sharing thoughts.

2.3.4 Static analysis

Static analysis is a validation and verification technique in which the software is analyzed without execution. There are two main approaches.

- **Pattern-based** static analysis checks basic static properties with error patterns, e.g. unused variable, ignored return value. Tools: *Coverity*, *FindBugs*, *SonarQube*.
- **Interpretation-based** static analysis checks dynamic properties, such as null pointer dereferencing, indexing out of bounds. Tools: *Infer*, *PolySpace*.

Frama C Frama C[1] is a set of tools that aim to analyse C source codes. Frama C assembles numerous analysis technique into a single framework. This cooperative approach lets static analysers build on the result computed by other component of the framework.

Using static analysis one can gather information about the source code without executing it. It can be a measurement by some metrics e.g. number of comments per line, depth of embedded control structures. A different stance takes a deeper look in the source code and tries to find dangerous constructions that are probably bugs. Latter, heuristic tools can't find all the bugs and sometimes they alarm the user in such cases when they find something which they think to be a bug, but it is not (these cases are called *false positives*).

Frama C uses different plug-ins for extensive functionality. Figure 2.4 shows an execution of metrics plug-in for the motivation example. Among others, it has value analysis, occurrence, scope plug-ins. These are not too interesting for the motivation example, they are useful in case of a large code base.

```
meres ubuntu ~ Desktop frama $ frama-c -metrics motivation_example.c
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no preprocessing)
[kernel] Parsing motivation_example.c (with preprocessing)
[metrics] Defined functions (1)
          ====================
           isDangerousSituation (0 call);

          Undefined functions (0)
          =======================


          'Extern' global variables (0)
          =============================


          Potential entry points (1)
          ==========================
           isDangerousSituation;

          Global metrics
          ==============
          Sloc = 18
          Decision point = 5
          Global variables = 0
          If = 5
          Loop = 0
          Goto = 6
          Assignment = 4
          Exit point = 1
          Function = 1
          Function call = 0
          Pointer dereferencing = 0
          Cyclomatic complexity = 6
```

Figure 2.4: Frama C metrics plug-in

In addition, Frama C supports annotation-based verification, it is based on ACSL (ANSI/ISO C Specification Language) [19]. Using this, preconditions, and the expectations can be given for a function. For example the code below shows ACSL for the motivation example. It says the code may have two behaviours, dangerous or safe. In case of dangerous, there are two variables greater, than the boundary value, the function shall return 1.

[1] `https://frama-c.com/`

In case of safe, there are two variables equal or less, than the boundary value, the function shall return 0.

```
/*@ behavior dangerous:
    assumes (temperature1 > boundaryTemperature && temperature2 > boundaryTemperature)
    || (temperature1 > boundaryTemperature && temperature3 > boundaryTemperature)
    || (temperature2 > boundaryTemperature && temperature3 > boundaryTemperature);
    ensures \result == 1;
  behavior safe:
    assumes (temperature1 <= boundaryTemperature && temperature2 <= boundaryTemperature)
    || (temperature1 <= boundaryTemperature && temperature3 <= boundaryTemperature)
    || (temperature2 <= boundaryTemperature && temperature3 <= boundaryTemperature);
    ensures \result == 0;
*/
```

Figure 2.5 shows the execution of WP plug-in, which is a weakest precondition calculus for ACSL annotations through C programs. Figure 2.5a shows the execution result for the correct implementation, and Figure 2.5b shows for the faulty implementation.

(a) Verifying ACSL for correct code **(b)** Verifying ACSL for faulty code

Figure 2.5: Frama C ACSL verification

2.3.5 Formal verification

The aim of formal verification is to prove or disprove the correctness of an algorithm or a system with respect to a formal specification or property, using formal methods.

Formal methods comprise of mathematical techniques, mostly discrete mathematics and mathematical logics. These are used for making and verifying the specification, plans (models), implementation and documentation of a hardware or software system.

The process of formal verification can be seen at Figure 2.6, where formal model is a low-level (e.g. *Kripke structures*, *Labeled Transition Systems*, *Kripke Transition Systems*) or higher-level (e.g. *Petri nets*) or engineering model, and formal requirements are precise requirements that can be checked automatically (e.g. *Propositional Linear Temporal Logic*, *Computation Tree Logic*, *Computation Tree Logic**).

There are some aspects, which makes the application of formal methods hard.

- Modeling the real world is hard, because there can be lack of information and assumptions can be imprecise. This problem is independent from formal methods, but still makes their usage hard.
- Special knowledge is needed from the users, for example mathematical models and their signatures. It is possible that engineering models cover this from the users.
- The method of verification and synthesis may be complex, algorithms have boundaries. User intervention may be necessary.
- It is only efficient for small models, because model checkers use exponential algorithms. But the efficiency of tools is increasing.

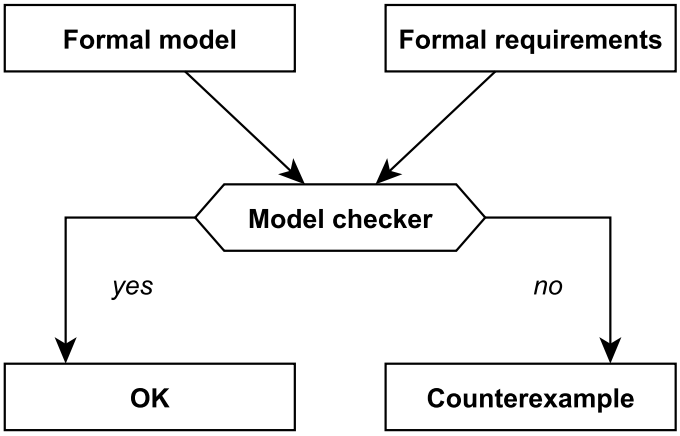


Figure 2.6: Process of formal verification

UPPAAL UPPAAL[2] is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata, extended with data types. Here, it is used to demonstrate formal verification on the motivation example.

First, a model has to be created, Figure 2.7 shows the created model. It first computes whether it is a safe or dangerous situation, then steps to the corresponding state. Then the model has to be instantiated, Figure 2.8 shows the used instances.

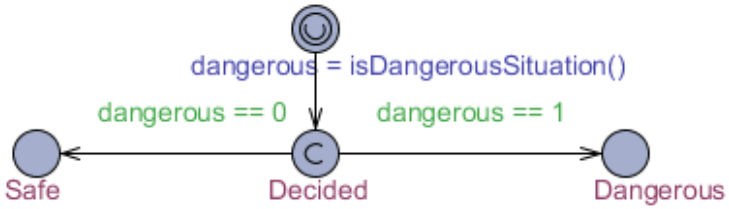


Figure 2.7: UPPAAL model

```
SafeAll = Decision(100, 99, 98, 97);
SafeFirstNot = Decision(100, 105, 98, 97);
SafeSecondNot = Decision(100, 99, 101, 97);
SafeThirdNot = Decision(100, 99, 98, 110);

DangerousFirstNot = Decision(100, 100, 102, 104);
DangerousSecondNot = Decision(100, 101, 98, 104);
DangerousThirdNot = Decision(100, 102, 103, 85);
DangerousAll = Decision(100, 101, 105, 110);

system SafeAll, SafeFirstNot, SafeSecondNot, SafeThirdNot, DangerousFirstNot, DangerousSecondNot,
     DangerousThirdNot, DangerousAll;
```

List 2.8: Instantiating the model

In the end, the instances can be verified with CTL (Compuational Tree Logic) expressions. Figure 2.9 shows the result of the verification for the correct implementation, Figure 2.10 shows for the faulty implementation. There are four CTL expressions on the figures. The first means, for the models with dangerous parameters, the model will reach Dangerous state in all possible execution path. The second means, for the models with dangerous

[2] http://www.uppaal.org/

parameters, all future state for all possible execution path will not be the Safe state. The third means, for the models with safe parameters, the model will reach Safe state in all possible execution path. The fourth means, for models with safe parameters, all future state for all possible execution path will not be the Dangerous state.

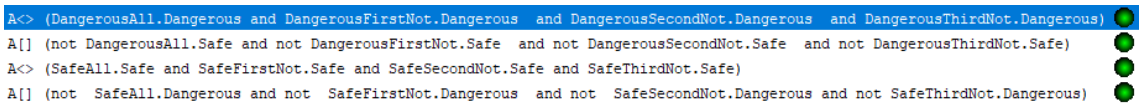


Figure 2.9: UPPAAL verification for correct implementation

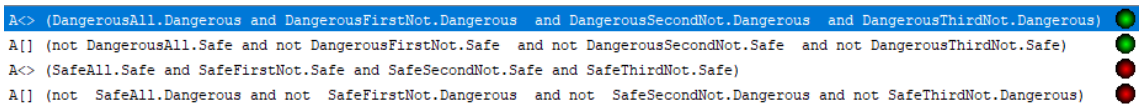


Figure 2.10: UPPAAL verification for faulty implementation

2.3.6 Testing

Testing is a dynamic software validation and verification technique, which means the program is being executed during testing, unlike with static analysis. Testing aims at discovering erroneous behaviour of the system. Unlike formal verification, testing cannot prove that a software is free of flaws, it can only show if it finds a bug. Although, testing cannot prove the absence of errors in a software, it is the most widely used technique in software verification. The reason for this is, that usually the failure of the program does not have so significant effect that it is worth to use other technique for verification, for example formal verification.

The basic elements of testing are test cases. Each test case is comprised of test inputs and an expected output. After the test case execution the actual output is compared to the expected. If they equal then that test case is said to be successful, it they are not equal then that test case fails. There is a third option, in case the test fails during execution, for example an unhandled exception occurs, the output of the test case is error.

Besides detecting defects, testing has a couple of other goals. Such as preventing defects, providing information about code quality and supporting decision making (e.g. about a release).

There are different classifications for testing, by several sights.

Scope of testing

- In case of unit testing the goal is to make sure that the basic elements of the program *(units)*, are functioning well. Usually unit tests are written by the developers. In TDD *(test driven development)* the unit tests are written first, then the implementation.
- In case of integration testing the goal is to make sure that the basic elements of the program can work together well, the communication is alright, and a given part of the system works properly.
- In case of system testing the whole program is being tested. The goal is to make sure that the program is functioning as it is expected.

- In case of regression testing, the initial situation is there are already successful tests. Then the code changes, new functionality is added, some older statement maybe modified. After that conviction is needed, that the old functionality is still working, the modifications did not break them.

Black-box testing

In case of black-box testing the source code and its structure is completely unknown for the tester. Tests can be built up by the specification and the requirements. The most common techniques in this case are equivalence classes, boundary values, decision tables and combinatorial testing.

Checking the specification with equivalence partitioning, two partition can be defined for the temperature values. First partition contains values lower, or equal to boundary temperature. Second partition contains values higher, than the boundary value.

Using boundary value analysis, three values should be used for each temperature: boundary value, boundary value - ε, boundary value + ε, where ε is an arbitrary little value, it depends on the type of the temperature (not the same for integer and float).

Test cases can be derived with, for example n-wise testing. In this case, three variables are used, and their possible values are also three. So, n-wise testing provides $3^3 = 27$ test cases.

White-box testing

In case of white-box testing the source code and its inner structure is known. Testing can be done based on internal behaviour of the program.

Test metrics Since testing cannot prove that there is no flaws in the software, it is an interesting question when to stop it. Test metrics are used for this purpose. They give feedback to the tester about how thoroughly the program has been tested. It is hard to find an indicator of quality testing, but the experience shows that it has correlation with coverage criterion. So the most used test metrics are coverage criterion.

Before introducing the coverage criterion, here is the explanation of some phrases used later.

- *Block*: a sequence of one or more consecutive executable statements containing no branches.
- *Condition*: a single logical expression, without any logical operator (and, or).
- *Decision*: a logical expression consisting of one or more conditions combined with logical operators.
- *Path*: a sequence of events, e.g. executable statements, of a component typically from an entry point to an exit point.

List of common coverage criterion:

- *Statement coverage:* the number of statements executed during test is divided by the number of all statements in the program. Figure 2.11 shows an example for statement coverage, if *a* is true, than both *x = 1* and *x++* will be executed.

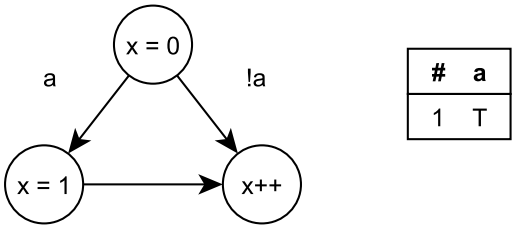


Figure 2.11: Statement coverage

- *Decision coverage:* outcomes of decisions taken during testing are divided by the number of all possible outcomes. It does not take into account all combinations of conditions. Figure 2.12 shows an example for decision coverage, both branches are taken, with the defined cases. Decision coverage does not imply condition coverage, it can be seen on the example, because both branches are taken, but b does not take true value.

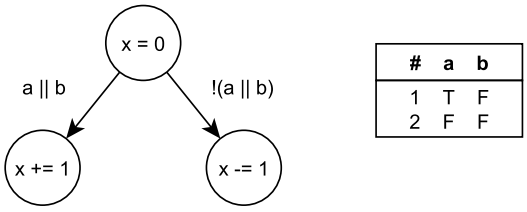


Figure 2.12: Decision coverage

- *Condition coverage:* number of tested combinations of conditions is divided by the number of aimed combinations of conditions. It does not yield 100% decision coverage in all case. Figure 2.13 shows an example for condition coverage.

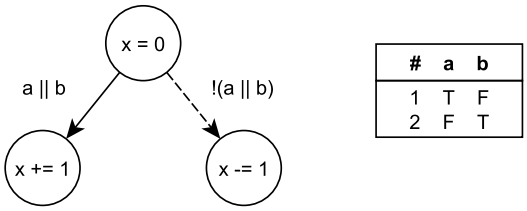
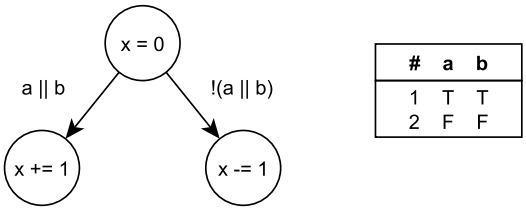


Figure 2.13: Condition coverage

- *Condition/Decision coverage (C/DC):* every decision has taken all possible outcomes at least once and every condition has taken all possible outcomes at least once. It does not take into account whether the condition has any effect. Figure 2.14 shows an example for C/DC. In case of $a == true$ and $b == true$, b does not have any effect on the decision, because it would always be true, as a is true.

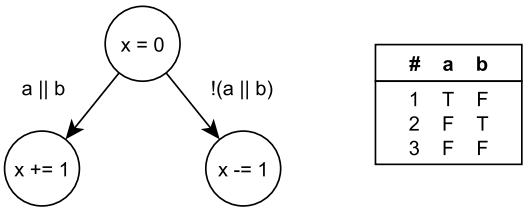
- *Modified Condition/Decision coverage (MC/DC):* Each entry and exit point has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, every decision in the program has taken all possible



#	a	b
1	T	T
2	F	F

Figure 2.14: Condition/Decision coverage

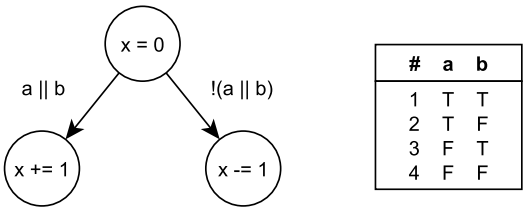
outcomes at least once, each condition in a decision is shown to independently affect the outcome of the decision. Figure 2.15 shows an example for MC/DC.



#	a	b
1	T	F
2	F	T
3	F	F

Figure 2.15: Modified Condition/Decision coverage

- *Multiple Condition coverage:* Every combinations of conditions are tried. Figure 2.16 shows an example for multiple condition coverage.



#	a	b
1	T	T
2	T	F
3	F	T
4	F	F

Figure 2.16: Multiple Condition coverage

- *Path coverage:* number of independent paths traversed during testing is divided by the number of all independent paths. Figure 2.17 shows an example for path coverage, it has two individual paths, both are taken with the specified cases.

2.4 Code-based test generation

Code-based test generation is obviously a white-box testing technique, because as its name says, it generates tests based on the code. Its advantage compared to manual test generation is the following. Creating tests manually can be cumbersome if the code base is large, furthermore the tester may omit important cases, because of human factors. Moreover if the source code is modified then the tester has to start this task again. It is much more simple to create the test automatically.

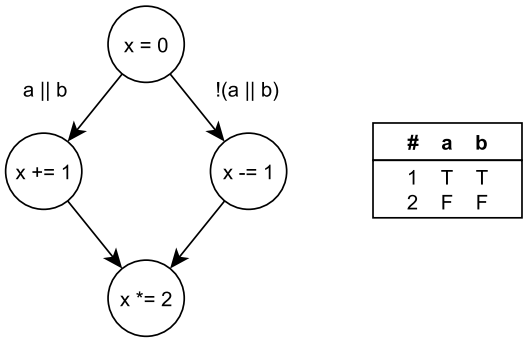


Figure 2.17: Path coverage

There are several method for code-based test generation e.g. *random generation*, *symbolic execution*, *model checking*, *annotation-based*. The following subsections present these methods.

2.4.1 Random generation

Random generation is a method, which uses, as its name says, random generating algorithm to obtain input values, for the test. It may sound simple, but there are many implementation of such a method.

Empirical researches shows, that the inputs causing errors tend to create a continuous (error) region. Consequently the inputs, which do not lead to error, have to create another continuous region. So if a previous test did not explore an error, the next test case has to be far from the previous one, this leads to an even distribution of test cases in the input range. This gives the basics of adaptive random testing [4], also known as ART. It seems, there are typical failure patterns: block pattern, strip pattern, and point pattern [8]. When the failure patterns are block or (thick) strip it is more likely that a further case will reveal a fault, than a near case.

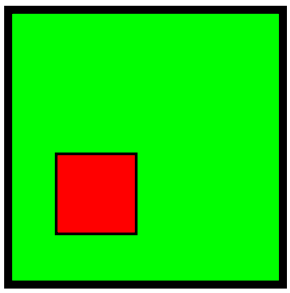
There are people who think adaptive random testing is not so good, because most papers which have been published in ART, work on either simulations or case studies with unreasonably high failure rates. On the other hand, ART is inefficient even on trivial problems when accounting for distance calculations among test cases, to an extent that probably prevents its practical use in most situations. For example, on the infamous Triangle Classification program, random testing finds failures in few milliseconds whereas ART execution time is prohibitive [5].

2.4.2 Symbolic execution

Symbolic execution is a program analysis technique, which parses the code, to automatically generate test data for the program. Symbolic execution uses symbolic values instead of concrete values for the input of the program. Variables are represented through the symbolic values of the inputs. In the process of symbolic execution at every moment the state of the executed program involves the symbolic values of the variables at the given point, and a path constraint, that tells how the given point is reachable, and lastly a program counter. The path constraint is a Boolean formula. To generate inputs from them,

```
int blockPattern(int x, int y, int) {
  if ((x >= 10 && x <= 12) &&
      (y >= 8 && y <= 11)) {
    return x / 2 * y;
    // it should be:
    // return x / 7 * y;
  }
  else {
    return x * y;
  }
}
```

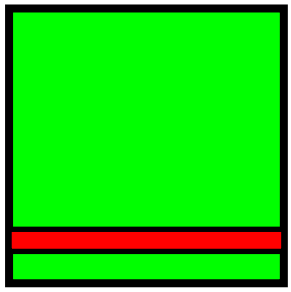
(a) Example code for block pattern



(b) Example figure for block pattern

```
int stripPattern(int x, int y) {
  // should be:
  // if (2 * x - y > 18)
  if (2 * x - y > 10) {
    return x / 2 * y;
  }
  else {
    return x * y;
  }
}
```

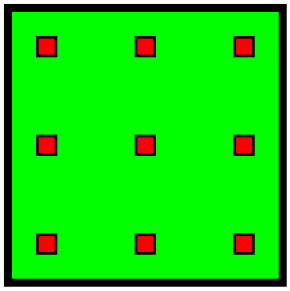
(a) Example code for strip pattern



(b) Example figure for strip pattern

```
int pointPattern(int x, int y) {
  if (x % 4 == 0 && y % 6 == 0) {
    return x / 2 * y;
    // should be:
    // return x / 7 * y;
  }
  else {
    return x * y;
  }
}
```

(a) Example code for point pattern



(b) Example figure for point pattern

an SMT (Satisfiability Modulo Theory) solver is used. The program counter tells the next statement. In most application the aim is to cover all possible execution path [12].

There are some problem with symbolic execution.

- *Path explosion:* it is hard to symbolically execute a large subset of all path, because most of the softwares contain vast amount of paths. It has exponential correlation.
- *Path branching:* real programs are often written in more than one programming language, or some parts are only available in binary form. It requires huge effort to compute precise constraints on these. If the path constraint is not precise, then path branching can occur. Meaning the path of the program during the execution with test data may differ form path, on which test data were generated.
- *Complex constraints:* it is possible, that a path constraint cannot be solved, because the general class of the constraint is not decidable.

2.4.3 Search-based

SBST (search-based software testing) is a branch of search-based software development. Searching algorithms are used to automate to process of finding test data, which maximizes the that test goals are reached, while minimizes the cost of testing.

SBST algorithms should be used for not only test input generation, but for generating test oracles and test cases as well. Most of the previous researches cared about test input generation, and output generation was in the background.

SBST and dynamic symbolic execution may work well together, because they could overcome each other's weaknesses [4].

2.4.4 Model checking

Model checkers work on some kind of model. So the source code has to be transformed to such a model, for example control-flow graph/automaton. Then model checkers are able to check reachability properties. This can be used to generate test inputs to reach arbitrary line of code.

A concrete model checking implementation is BMC (bounded model checking). Its main idea is to check the properties bounded by the length of the paths, instead of handling the state space all together. This way the state space is traversed iteratively, each time it only goes to the bounds. This boundary can be estimated by intuition of the problem's size or by the worst case execution time. For example CBMC (C Bounded Model Checker) uses this kind of approach.

2.4.5 Annotation-based

If the code contains pre- and post-conditions or other annotations, these are able to guide the test generation. A similar tool is QuickCheck, which uses property-based testing. The programmer has to write assertions about logical properties that the function has to fulfill. Then it attempts to generate a test-case that falsifies these assertions.

The ACSL, used by Frama C, mentioned in the previous section also belongs here.

My subjective opinion is annotation-based approach can be the best from the mentioned methods. I think the other methods are good for generating inputs for example for a function, but lack the ability to tell anything about the output of that function, while annotation-based generation is capable of telling whether the output is correct for that input or not. On the other hand, it requires correct annotations on a function, which has to be written by somebody. I think writing this kind of annotations cannot be automated, so it requires the developer or tester to write it, and importantly not based on the code, but based on the specification. Maybe in the (near) future artificial intelligence will be able to generate annotations from the specification. So the annotations can also be faulty, which is the same case as in model-based testing creating the model.

Chapter 3

Test generating tools

In this chapter, some existing code-based test generating tools are introduced. Their functioning is presented with the motivation example. I had tried these tools before the own tool was designed, so the experiences gained from these tools could be used.

3.1 CREST

CREST[1] is an automatic test generation tool for C. CREST works by inserting instrumentation code into a target program to perform symbolic execution concurrently with the concrete execution, this is called concolic testing. The generated symbolic constraints are solved to generate input that drive the test execution down new, unexplored program paths. CREST can be guided not only with depth-first search strategy, but also by the control flow graphs of the program under test [7]. It depends on Yices (an SMT solver) and CIL (C Intermediate Language). CREST is no longer being actively developed.

To run CREST, the following codes have to be inserted into the motivation example.

```
#include <crest.h>
```

```
int main() {
  int t1, t2, t3, bt;
  CREST_int(t1);
  CREST_int(t2);
  CREST_int(t3);
  CREST_int(bt);

  isDangerousSituation(bt, t1, t2, t3);
}
```

CREST_int(t1) is used to make the t1 variable symbolic. First *bin/crestc* has to be executed, which instruments and compiles the program. Then CREST can be run on the instrumented program, with the

```
run_crest PROGRAM NUM_ITERATIONS -strategy
```

command, where strategy can be dfs, cfg, random, uniform_random, random_input.

The motivation example was run with

```
./bin/run_crest  ./test/motivation_example 10 -dfs
```

command. Figure 3.1 shows the execution result of the command.

[1] https://github.com/jburnim/crest

```
Iteration 0 (0s): covered 0 branches [0 reach funs, 0 reach branches].
Iteration 1 (0s): covered 2 branches [1 reach funs, 10 reach branches].
Iteration 2 (0s): covered 5 branches [1 reach funs, 10 reach branches].
Iteration 3 (0s): covered 6 branches [1 reach funs, 10 reach branches].
Iteration 4 (0s): covered 7 branches [1 reach funs, 10 reach branches].
Iteration 5 (0s): covered 9 branches [1 reach funs, 10 reach branches].
Iteration 6 (0s): covered 10 branches [1 reach funs, 10 reach branches].
```

Figure 3.1: Result of running CREST on motivation example

My opinion is the output of CREST is not meaningful, the tester does not get information about the code.

3.2 FShell

FShell[2] is a testing tool for C programs, which has an own language called **FQL** (FShell Query Language). This is a simple, concise language with precise semantics, moreover it helps the engines to generate test cases and compute achieved coverage. FShell is based on CBMC (C Bounded Model Checker) and it uses CFA (Control Flow Automata) to represent the program [13].

FShell has an own command line, in which the user can define his/her desired queries. The program under test has to be granted when the user starts FShell. At this point the user can define *unwinding* parameter, which tells FShell, how many steps of the loops has to be unrolled.

This is the syntax of FShell queries: **in** *scope* **cover** *goals* **passing** *constraints*

- *scope*: CFA transformer/filter function
- *goals*: coverage pattern
- *constraints*: path pattern

Here are some example queries:

```
cover @3                              // Cover 3rd line
cover @CONDITIONEDGE                  // Evaluate atomic conditions to both true and false
cover @DECISIONEDGE                   // Evaluate conditional statements to both true and false
cover @CONDITIONEDGE+@DECISIONEDGE    // Union of condition and decision coverage
```

If a query cannot be complied, then FShell returns a counterexample, else it returns test cases that contain the input values for the variables.

Motivation example has to be extended with a main function, but here the variables were not made symbolic.

```
int main() {
  int temperature1;
  int temperature2;
  int temperature3;
  int boundaryTemperature;
  isDangerousSituation(boundaryTemperature,
    temperature1,temperature2, temperature3);
}
```

After entering the shell *cover @conditionedge* command generated the output seen at Figure 3.2. It generated values for the inputs of the function under test. There are cases for both dangerous and safe decisions.

[2] https://forsyte.at/software/fshell/

```
IN:
boundaryTemperature=1
temperature1=2
temperature2=536870912
temperature3=0

IN:
boundaryTemperature=1
temperature1=2
temperature2=-1610612736
temperature3=536870912

IN:
boundaryTemperature=5
temperature1=2
temperature2=-1610612732
temperature3=536870916
```

```
IN:
boundaryTemperature=5
temperature1=2
temperature2=536870916
temperature3=-1610612732

IN:
boundaryTemperature=-2147483643
temperature1=-2147483646
temperature2=-1610612732
temperature3=536870916

IN:
boundaryTemperature=-1610612735
temperature1=-1610612730
temperature2=-2147483648
temperature3=-2147483648
```

Figure 3.2: FShell output for @conditionedge coverage

3.3 KLEE

KLEE is a symbolic virtual machine, capable of automatically generating test inputs for complex programs such as device drivers, network drivers, and utility programs written in C [6]. It is built on the infrastructure of compiler LLVM. It is available as docker image. The parameters of the tested function has to be made symbolic by hand.

```
klee_make_symbolic(&a, sizeof(a), "a");
```

Using KLEE is a bit complicated. The variables have to be made symbolic, like in the case of CREST. KLEE cannot handle C program, where the main function does not have an int and a char** argument.

To run KLEE, a docker container has to be created and run.

```
docker run -ti --name=klee_container --ulimit='stack=-1:-1' klee/klee
```

Somehow the code under test has to be copied inside the container. It is practical to use *echo "content of the .c file" > motivation.c*, at default only vi is installed in the container. Test generation can be executed with the following commands.

```
clang -emit-llvm -g -c motivation.c -o motivation.bc
klee --libc=uclibc --posix-runtime motivation.bc
```

After this, a new folder *klee-last* contains the test cases. These test cases are not in textual format, so to see their content, the following command has to be used.

```
ktest-tool --write-ints klee-last/test000001.ktest
```

The generated test for the motivation example by KLEE are presented at Figure 3.3.

3.4 PathCrawler

The main functionality of PathCrawler is automating the structural unit testing, by generating test inputs that covers every possible execution path of the tested function.

It has an online surface, where it can be tried. It is possible to upload own code, but it has built-in examples. Test settings can be modified, for example the range of variables, prerequisites, strategy. In addition, the tester can specify a test oracle.

```
ktest file : 'test000001.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: 0
object    2: name: b't2'
object    2: size: 4
object    2: data: 0
object    3: name: b't3'
object    3: size: 4
object    3: data: 0
object    4: name: b'bt'
object    4: size: 4
object    4: data: 0

ktest file : 'test000002.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: 0
object    2: name: b't2'
object    2: size: 4
object    2: data: 805306368
object    3: name: b't3'
object    3: size: 4
object    3: data: 0
object    4: name: b'bt'
object    4: size: 4
object    4: data: 536870912

ktest file : 'test000003.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: -2080374784
object    2: name: b't2'
object    2: size: 4
object    2: data: 0
object    3: name: b't3'
object    3: size: 4
object    3: data: 0
object    4: name: b'bt'
object    4: size: 4
object    4: data: -2013265920
```

```
ktest file : 'test000004.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: 134217728
object    2: name: b't2'
object    2: size: 4
object    2: data: 0
object    3: name: b't3'
object    3: size: 4
object    3: data: 256
object    4: name: b'bt'
object    4: size: 4
object    4: data: 0

ktest file : 'test000005.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: 0
object    2: name: b't2'
object    2: size: 4
object    2: data: 0
object    3: name: b't3'
object    3: size: 4
object    3: data: 0
object    4: name: b'bt'
object    4: size: 4
object    4: data: -1073741824

ktest file : 'test000006.ktest'
args       : ['motivation.bc']
num objects: 5
object    0: name: b'model_version'
object    0: size: 4
object    0: data: 1
object    1: name: b't1'
object    1: size: 4
object    1: data: 1
object    2: name: b't2'
object    2: size: 4
object    2: data: 0
object    3: name: b't3'
object    3: size: 4
object    3: data: 0
object    4: name: b'bt'
object    4: size: 4
object    4: data: 0
```

Figure 3.3: Tests generated by KLEE

The output of PathCrawler is clear, it can help a tester to see the different cases, and check if everything went well. To the right, one test case can be found, at Figure 3.4 the online surface. It generated six test cases. Inputs, outputs verdicts, path predicate can be examined for all cases.

```
boundaryTemperature=0
temperature1=1078660954
temperature2=0
temperature3=0
output=0
```

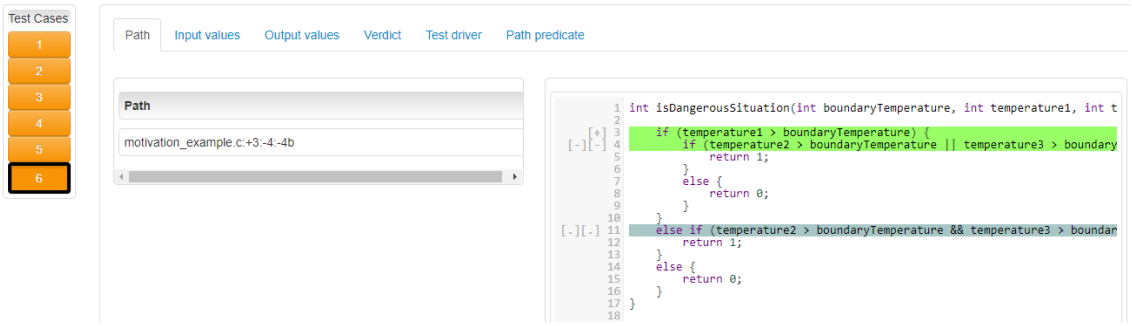


Figure 3.4: Test cases generated by PathCrawler

3.5 Evaluation

Table 3.1 shows the comparison of the tools. The first aspects of the comparison are the different coverage metrics. The second aspects are the used techniques for the test generation. The third aspects are outputs for the tester.

Aspect	CREST	FShell	KLEE	PathCrawler
Statement coverage	yes	yes	yes	yes
Decision coverage	yes	yes	yes	yes
Condition coverage	yes	yes	yes	yes
MC/DC	yes	yes	no	yes
Reaching given line	no	yes	no	no
Symbolic execution	yes	no	yes	yes
Model checking	no	yes	no	no
Random	yes	no	yes	no
Input generation	no	yes	yes	yes
Saving output values	no	yes	yes	yes
User defined test oracle	no	no	no	yes
Generating executable code	no	no	no	no
Showing coverage metrics	no	no	no	yes

Table 3.1: Comparison of test tools

It seems like PathCrawler is the most usable tool from the examined tools. Though FShell's expressive query language is a nice thing.

My experience is CREST and KLEE are cumbersome to use, the tester has to make variables symbolic manually, and the generated output are hard to read. So, I want to create a more usable tool, mostly by the abilities of PathCrawler, extended with executable code generation.

Chapter 4

Design

This chapter introduces the design decisions of the thesis. This includes the design of use cases, architecture, and the selection of other used tools.

First of all, I decided to create an own tool, and not to extend, for example an existing tool with a plug-in. On the one hand, I wanted to design a tool from scratch, and get experience with it. On the other hand, I liked ACSL, so I was thinking about a plug-in for Frama C, which could generate ACSL for a given function, but it did not seem to be feasible. So, creating an own tool for test generation became the choice.

4.1 Use cases

Figure 4.1 shows the use cases of the tool. The tester is able to chose the desired use case with command line arguments.

- **Use case 1** - *Test the code*: this is the main use case, this includes all the others. *Test the code* means *Generate test*, then *Execute test*, finally *Evaluate tests*.
- **Use case 2** - *Generate tests*: its output is runnable C test code with the help of a C unit testing framework.
- **Use case 3** - *Execute tests*: during test execution the generated test code and the code under test are compiled, linked and executed. Its output is a test result. Obviously, it requires generation of the tests.
- **Use case 4** - *Evaluate tests*: creates document about coverage metrics of the original code. Prior to evaluation, test generation and execution both have to be run.

The ability of running the generation phase separately may be useful when the tester only wants to see what kind of tests are generated, or desires to use other compiler than what the tool uses. The tester can change his/her mind and decide to run the tests anyway, which implies the tester should also be able to run and evaluate the tests.

4.2 Architecture

Figure 4.2 shows the components of the tool. The components outside the the tool are quite concrete. They have been chosen for a good reason, it is detaild in the Section 4.3.

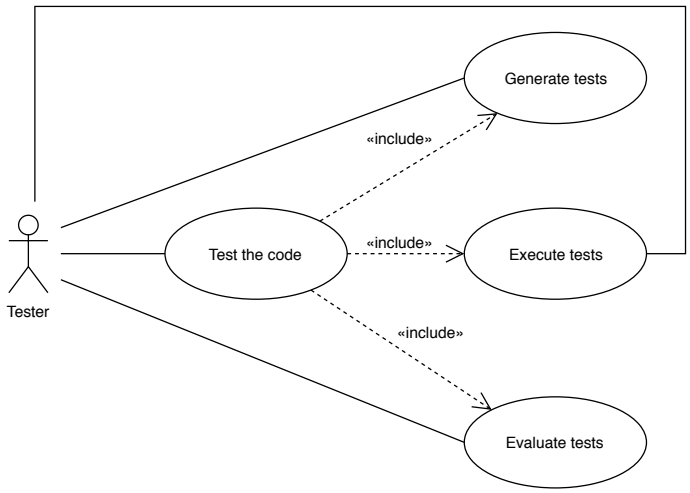


Figure 4.1: The use cases of the tool

CTestGenerator is the main component, it is the tool under development. It needs services from other components. **ANTRL** is needed by **Instrumenter** component, which uses the original C source code, the code under test. Instrumenter component requires gcc-s preprocessing service, it creates an instrumented version of the original code. The instrumented code is used by the **Backend** component, which requires services of **CBMC**. **TestFramework** component requires data from the Backend, using that, it creates a test file, **CMake** file, and then executes them. Finally **TestEvaluation** component creates html report of the test, using **Gcovr**, the test code and the original code.

Figure 4.3 shows the sequence diagrams of the tool's main functionality. **App** is the entry point of the program, it is called with command line arguments, which is processed by the **ArgumentParser**. Since the arguments define the desired functionality, it is the ArgumentParser's duty to create a convenient **TestStrategy**. Then, this TestStrategy is returned for the App, and App calls its *execute()* method. TestStrategy will call the appropriate components, in the right order. The called components depend on, the actual use case. Figure 4.3 represents the first use case (test the code), each component is involved in this.

4.3 Selection of tools

It was crucial at design phase to find appropriate tool for each step in the designed tool. Obviously, if there hadn't been any, then the design of the tool would have changed significantly, because writing a model checker or a language recognition tool is not a simple task.

In this section the selected tools are introduced, and the reason, why they were chosen.

4.3.1 ANTLR

ANTLR[1] is a tool for language recognition. It can be used to design context free grammars. Furthermore ANTLR supports the generation of lexer and parser for the language, so the

[1] `https://www.antlr.org/`

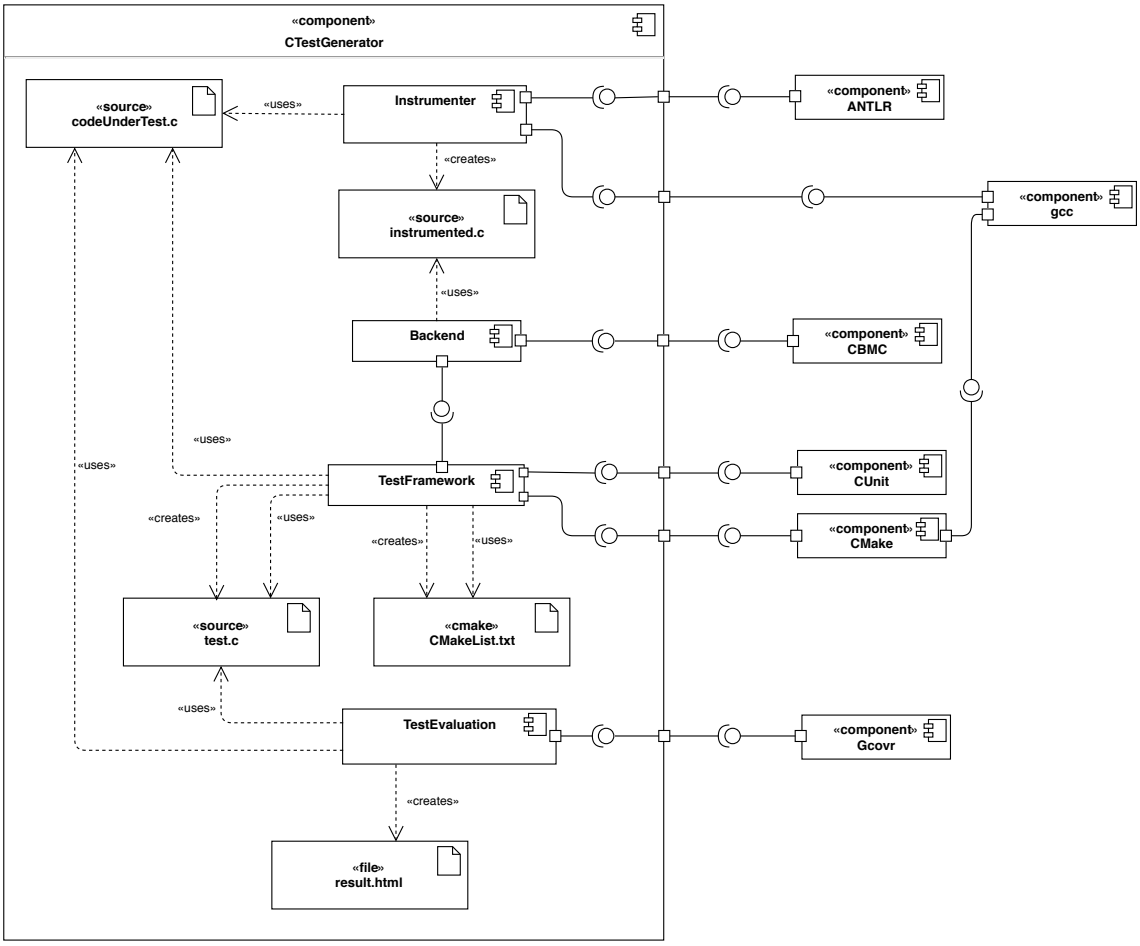


Figure 4.2: Components of the tool

user doesn't have to do that. In addition it is able to generate listener or visitor classes for the language. It facilitates the processing of the input.

C language is quite complex, writing the grammar for it could be a whole thesis work, so an existing grammar is used, which can be found at ANTLR's GitHub repository.[2]

ANTLR executes lexical analysis on the input code, which creates a token stream. Here is an example for its functioning. The input is $x = y + 10$, which is just a stream of characters. The output of the lexical analysis is five tokens: ⟨int⟩, ⟨identifier 1⟩, ⟨=⟩, ⟨identifier 2⟩, ⟨10⟩ [3]. After lexical analysis comes the syntax analysis. At this phase ANTLR builds a parse tree from the tokens, and denotes syntax errors, based on the grammar. Figure 4.5 shows the parse tree of a Program 4.4. This parse tree is ours, we can do whatever we want to do with it. For example visiting the tree with the help of the generated visitor is a good idea. Section 5.2 details what is being done with parse trees in the tool.

Why ANTLR? I chose ANTLR, because it was obvious that I need tool for code instrumentation. It is difficult to tell what should be instrumented. A simple solution is to instrument each line, but it is possible to write the whole program in just one line, although it is considered a bad practise. Another simple idea is to instrument after semi-column, but there can be cases when it is not practical to write code after semi-column, for

[2] `https://github.com/antlr/grammars-v4/blob/master/c/C.g4`

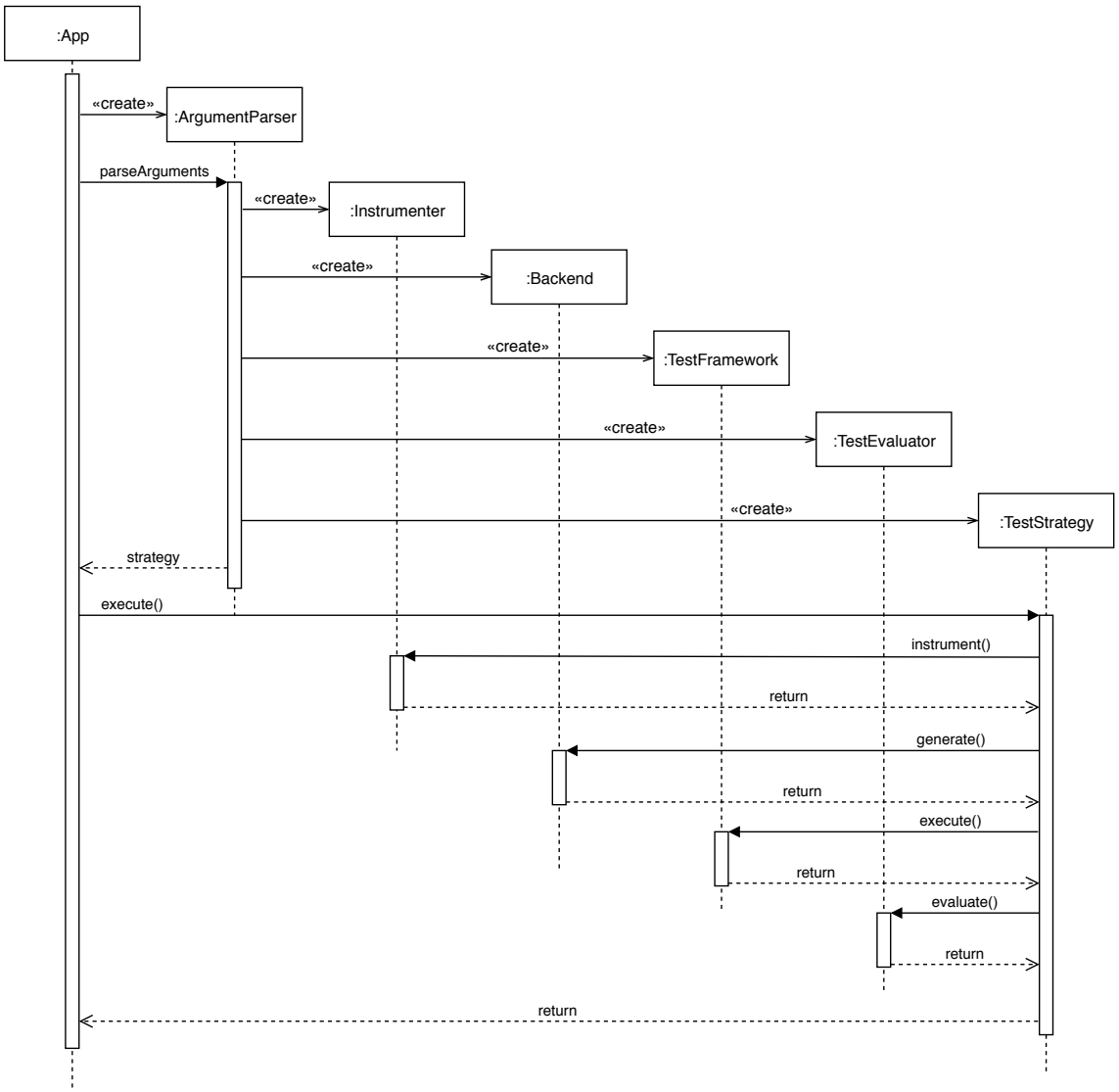


Figure 4.3: Sequence diagram of the main functionality

```
int main() {
  printf("Hello, World!");
  return 0;
}
```

Program 4.4: Hello world program

example after function declarations, return statements; and the first statement is ignored by this solution. So, it is not impossible to do this way, but I am sure it is very hard to cover each case, for each valid C program. Language recognition tools are invented to solve this program, why not to use them?

As a matter of fact, I have some experience with ANTLR. Although, I also have experience with Xtext, which is a similar tool to ANTLR, but Xtext works as an Eclipse plug-in and generates Eclipse plug-in, I wanted an individual tool, so Xtext got out of the possible choices.

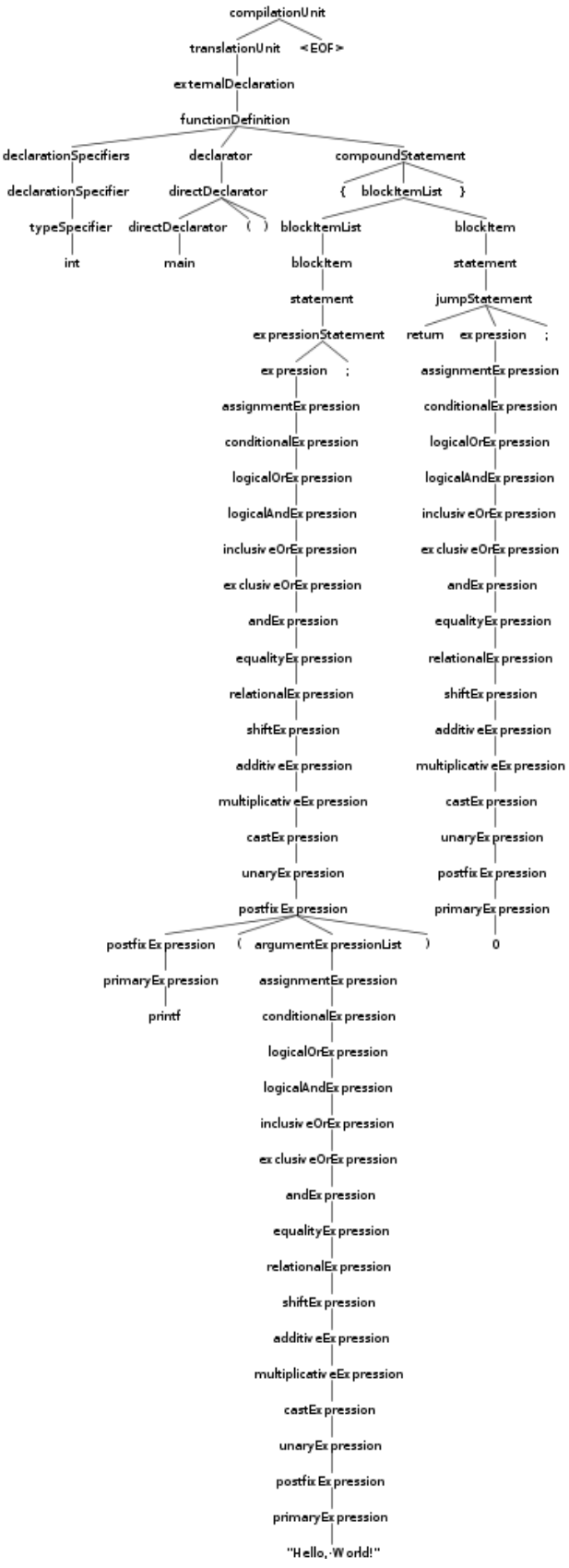


Figure 4.5: Hello world parse tree

4.3.2 CBMC

ANSI-C Bounded Model Checker[3] is a tool that can formally verify ANSI-C programs. This includes pointer safety, array bounds, and user-provided assertions [9]. The plan was to use CBMC in the tool as a test input generator. At instrumentation phase assert statements are placed, and CBMC returns input values with which those asserts can be reached. This can be used to cover each line and branch.

Why CBMC? The first plan was to use Theta, a model checking framework, developed inside the research group. It would have been practical for me, because the developers are there at the university and if I has any question I can get answers and help quickly. Also, it would have been practical for them, because their tool would have one more application, raising its value. Unfortunately, it supports C programs only in the form of CFA (control flow automata), and there wasn't any trustworthy tool for translation of C to CFA. Implementing such a tool seemed to be a bit exaggerate. Because of that, I looked up if there is any tool for this, and I found CBMC.

4.3.3 CUnit

CUnit[4] is as unit testing framework for C, it is capable of writing, administering and running unit tests for C. It is built as a static library, which has to be linked to the test code.

Why CUnit? I was searching for a unit test framework and it was one of the first result, given by Google. I tried it, it worked, and looked lightweight, so I chose this one. Later in the document more possibilities are listed, which may be included in the tool in the future.

4.3.4 CMake

CMake[5] is build tool. Its great advantage, it uses simple platform and compiler independent configuration files, and generates native makefiles and workspaces that can be used with the desired compiler environment.

Why CMake? At the beginning I tried to forge gcc command by myself. It didn't work out well, so I tried to generate Makefile for compilation. It was still too low level, too many information was needed. Finally, I tried CMake, that was high level enough for my needs.

4.3.5 Gcovr

Gcovr[6] is a tool that generates summarized code coverage results. This is inspired by the Python coverage.py package, which provides a similar utility in Python.

[3] `https://www.cprover.org/cbmc/`
[4] `http://cunit.sourceforge.net/`
[5] `https://cmake.org/`
[6] `https://gcovr.com/guide.html#getting-started`

Why Gcovr? It was easy to install and worked for the first time, so I liked it.

4.4 Summary

In this chapter the use cases and the architecture of the tool have been presented. The main use cases are test generation, test execution, test evaluation, and testing the code, which involves all prior use cases.

Moreover, the selection of the dependencies have been detailed. These dependencies are ANTLR, CBMC, CUnit, CMake, and Gcovr.

Chapter 5

Implementation

This chapter details the implementation of the tool. Here comes some general information, which is followed by a section for each major step of the tool.

5.1 Overview

The tool is made with Java, using Maven as the project management tool. Maven uses an xml file called project object model (POM.xml), in which one can define the dependencies and build informations of the project. It is useful for handling dependencies, because the only task is to write a few lines, then Maven downloads the dependency automatically from Maven central repository, or if it is already downloaded, then looks for it in the local repository. It is also useful for building, Maven has numerous phases, e.g. *compile* (compiles the source code), *test* (tests the compiled source code using a suitable unit testing framework), and *package* (packages the compiled code in its distributable format, such as jar).

Dependencies of the tool

- **ANTLR** - serves as helper for instrumentation, this has been mentioned earlier and will be mentioned later as well.
- **JUnit** - serves as the unit testing framework, used for unit testing some functions of the implementation.
- **slf4j** - serves as a simple facade for various logging frameworks.
- **log4j** - serves as the logging framework used in the tool.
- **json-simple** - is used for parsing and forging JSON files.
- **Apache Commons** - is used for parsing command line arguments.

Command line options

- **General options**

  `-r --root`: absolute path to the project's root folder

`-in --input`: name of the .c file under test

`-l --link`: libraries to be linked

`-h --help`: print help

- **Functionality options**

  `-gen --generate`: generate tests

  `-exe --execute`: execute tests

  `-eval --evaluate`: evaluate tests

- **Tool options**

  `-it --instrumentation-tool`: select instrumentation tool from the list: antlr (default)

  `-gt --generation-tool`: select backend tool from the list: cbmc (default)

  `-tf --test-framework`: select test framework from the list: cunit (default)

  `-et --evaluation-tool`: select evaluation tool from the list: gcovr (default)

- **Instrumentation options**

  `-f --function`: list of functions under test (default is all function)

  `-c --called`: not only test functions specified with -f, but the ones called from them (recursively)

  `-mcdc`: create MC/DC coverage

The tester can choose functionality with command line options. To handle it simply in the implementation, strategy pattern is used. Figure 5.1 shows a class diagram containing App, ArgumentParser and Strategy classes. Its advantage is App doesn't need to know which strategy is being executed. ArgumentParser instantiates and returns a strategy based on the command line arguments, and App can call its execute() method, without caring about its concrete type. It can be noticed that the concrete strategies are paired with the use cases, except DoNothingStrategy, it is a dummy type, it is returned by the ArgumentParser if any fault occurs.

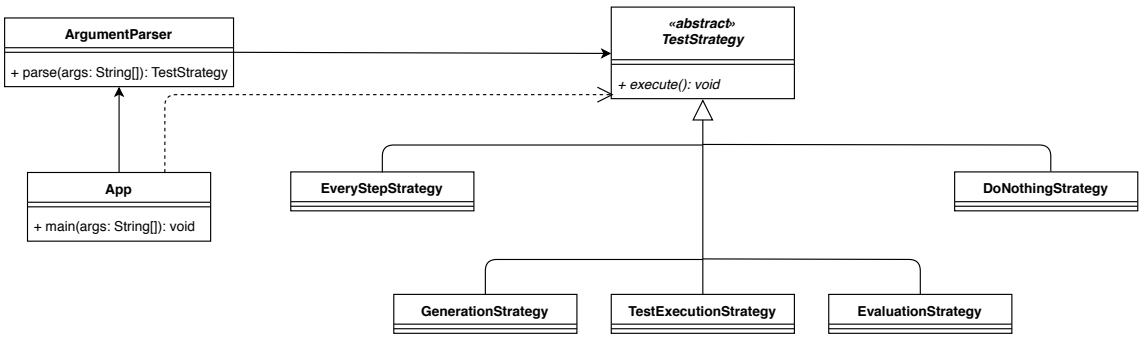


Figure 5.1: Class diagram of strategies

Required folder structure
Figure 5.2 shows the required folder structure. The tester should put *.c files into the src folder and *.h files into the include folder. The other folders and the CMakeLists.txt file are created by the tool. The testgen folder has four subfolders. Subfolder instrumentation contains files created at instrumentation phase, such as a preprocessed file, includes, and the final instrumented code, as well it contains the output files of the backend. Subfolder src contains the same *.c files as root/src and the test.c file, this way it simpler to compile the code. Subfolder evaluation contains the result of the tests, and the coverage result; the prior is txt file, the latter is html files. Folder build is also created by the tool, CMake is run from this folder, so every compilation file is in this folder. The CMakeLists.txt file is needed by CMake, it contains information for the compilation.

```
+build
+src
  -code_under_test.c
  -optionally_other_code.c
+include
  -header.h
  -header2.h
  -... .h
  -headerN.h
+testgen
  -instrumentation
  -src
  -evaluation
  -temp
+CMakeLists.txt
```

Figure 5.2: Required folder structure

Figure 5.3 presents the big picture of the tool. The workflow goes like this:

1. At the beginning there is some code that the tester wants to test. The tool is able to test only one .c file at a time, to test more, the tester has to call the tool again(, and save every output before running again).

2. At instrumentation phase, the code is being expanded with *assert* statements.

3. Then the backend generates test input to reach those assert statements.

4. The generated test inputs are used to create C unit tests.

5. The generated test file and the original source code are compiled, executed and evaluated.

6. Test result and coverage result are generated.

5.2 Instrumentation

At the beginning, the plan was to use instrumenter component only for inserting assert statements before each statement in the source code. It turned out, it can be, and should be used for more than that. With it, the called functions from a function can be defined, MC/DC transformation can be done, and function headers can be obtained, which will be needed later on.

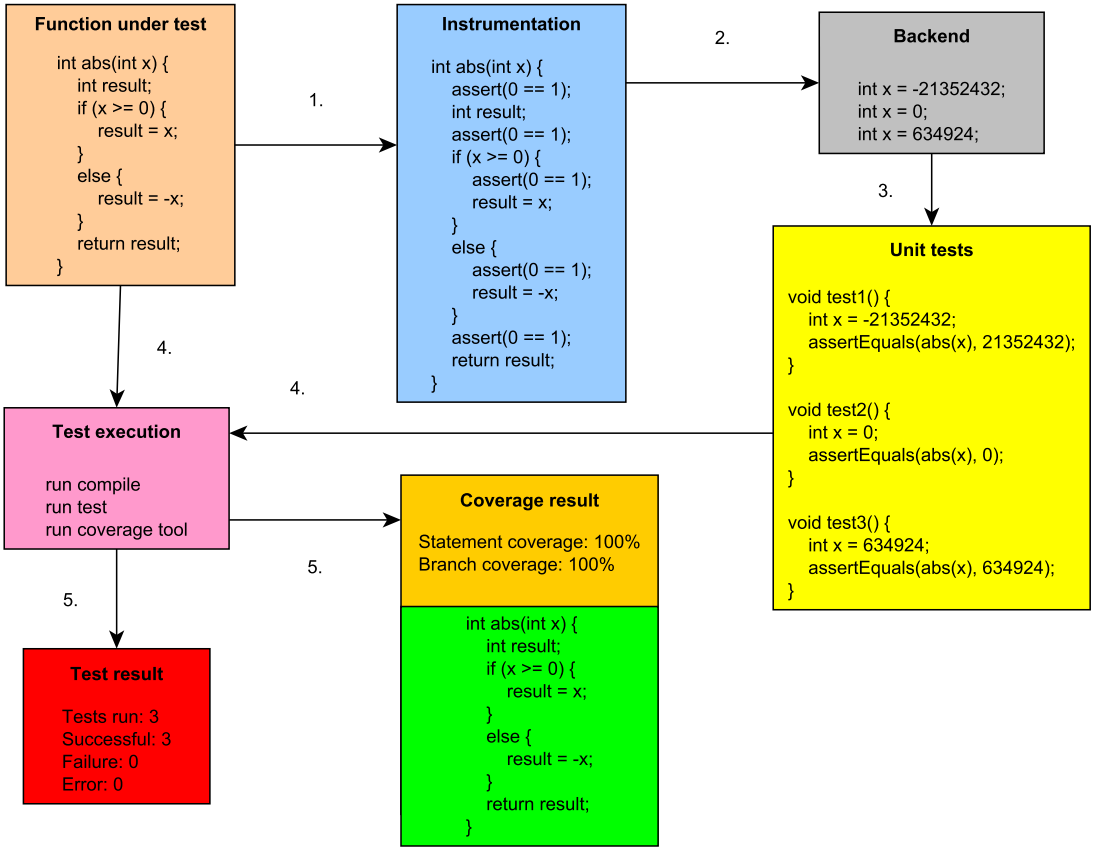


Figure 5.3: Tool's workflow

As it was mentioned a couple of times earlier, ANTLR is used for aiding the instrumentation. A significant advantage was, that the grammar was already written to C. Below a small fragment can be seen of the grammar; two parser rule: *selectionStatement* and *iterationStatement*.

```
selectionStatement
  :   'if' '(' expression ')' statement ('else' statement)?
  |   'switch' '(' expression ')' statement
  ;

iterationStatement
  :   While '(' expression ')' statement
  |   Do statement While '(' expression ')' ';'
  |   For '(' forCondition ')' statement
  ;
```

ANTLR generates a parse tree, from the source code, an example has occurred in the design chapter. ANTLR also generates a visitor, which can be overriden with own visitors. Figure 5.4 depicts the hierarchy of the visitor classes. The ancestor class is CBaseVisitor, it is generated by ANTLR, it doesn't really do useful things, just visits the tree. FunctionDefinitionCreator during visiting, creates FunctionHeader objects and store them into FunctionHeaders singleton class. ReturnCalledFunctionsFromGivenFunctionVisitor collects the names of functions, which was called from some given functions, and does this recursively, until no more function is added to the list. For example foo calls bar and bar calls foo2, then starting from foo, the result will be foo, bar and foo2 as well. This is needed if the user sets `-c` flag. These classes don't create any code, they just analyse it.

CCodeBaseVisitor writes the code without inserting any plus code inside it, its main purpose is to stand as a superclass for different instrumenting visitors. AssertGivenFunctionVisitor puts assert statements inside the given functions, while AssertEverythingVisitor puts assert into each and every function. It may be a good observation to note, that asserting everything could be done by calling AssertGivenFunctionVisitor with all function names, so AssertEverythingVisitor seems to be useless. But it is used, because if the user doesn't specify functions to instrument, then it is easier to visit the tree just once and instrument the code, than visiting the tree with each different function. MCDCTransformator doesn't instrument the code it resolves compound condition expressions into single ones. When instrumenting one has to be very careful, because selection and iteration statements may omit braces, if they contain only one statement. If this is the case, inserting assert before that statement would totally modify the functionality of the code, so during instrumentation {} are inserted as well, where they are necessary.

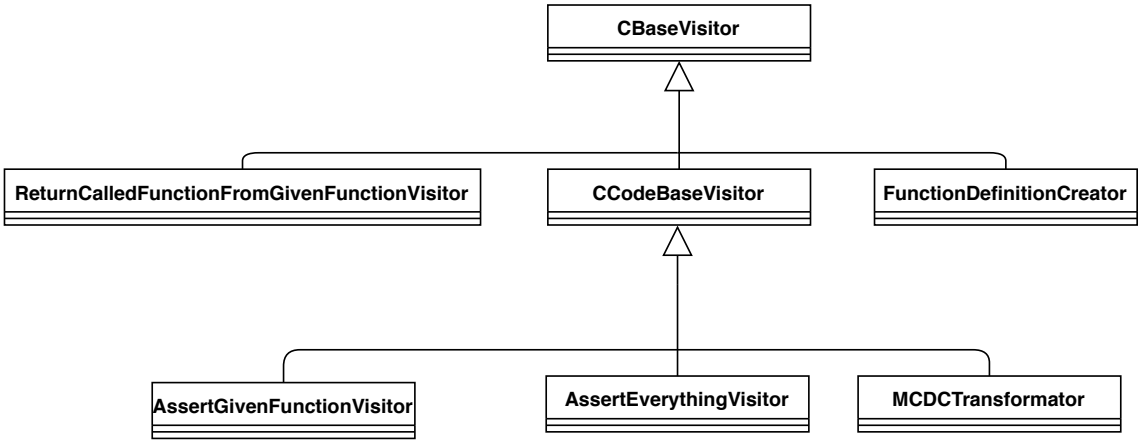


Figure 5.4: Hierarchy of visitor classes

5.2.1 MC/DC transformation

The aim of this transformation is to make the backend achieve hundred percent MC/DC coverage on the code. Figure 5.5 shows the transformation for OR expression, Figure 5.6 shows for AND expression, and Figure 5.7 shows the case, when there is negation in the expression. To handle negation, De Morgan's laws have to be used.

De Morgan's laws:

1. $\neg(P \vee Q) \iff \neg P \wedge \neg Q$
2. $\neg(P \wedge Q) \iff \neg P \vee \neg Q$

where $\neg$ is the *negation* logic operator, $\vee$ is the logical *or* operator, $\wedge$ is the logical *and* operator, $\iff$ stands for *equivalence*.

How does it help to reach hundred percent MC/DC coverage? Take a look at Figure 5.5a! If asserts are placed inside that function, then the test generator can cover them with two cases; for example a = 0, b = 0 and a = 1, b = 1. Now take a look at Figure 5.5b! The previously generated cases won't cover every assertion, the else if branch is intact, because it will either return from the if branch, or the else branch. This transformation makes sure that every condition's output depends the decision independently. This stands for the and expression as well, the negation doesn't matter in this, it just reverses decisions.

```
int anyGreaterThanZero(int a, int b) {
  if (a > 0 || b > 0) {
    return 1;
  }
  else {
    return 0;
  }
}
```

(a) C program snippet, with two conditions and an *or* expression.

```
int anyGreaterThanZero(int a, int b) {
  if (a > 0) {
    return 1;
  }
  else if (b > 0) {
    return 1;
  }
  else {
    return 0;
  }
}
```

(b) The program in **(a)**, after transformation.

Figure 5.5: *OR* transformation example

```
int bothGreaterThanZero(int a, int b) {
  if (a > 0 && b > 0) {
    return 1;
  }
  else {
    return 0;
  }
}
```

(a) A C program snippet, with two conditions and an *and* expression.

```
int bothGreaterThanZero(int a, int b) {
  if (a > 0) {
    if (b > 0) {
      return 1;
    }
    else {
      return 0;
    }
  }
  else {
    return 0;
  }
}
```

(b) The program in **(a)**, after transformation.

Figure 5.6: *AND* transformation example

```
int noneGreaterThanZero(int a, int b) {
  if (!(a > 0 && b > 0)) {
    return 1;
  }
  else {
    return 0;
  }
}
```

(a) A C program snippet, with a negation of an *and* expression.

```
int noneGreaterThanZero(int a, int b) {
  if (!(a > 0)) {
    return 1;
  }
  else if (!(b > 0)) {
    return 1;
  }
  }
  else {
    return 0;
  }
}
```

(b) The program in **(a)**, after transformation.

Figure 5.7: *De Morgan* transformation example

5.2.2 Challenges

Although, it was an interesting part, and I enjoyed creating the instrumenter component, it wasn't always easy. Starting it went quite bad, it was really hard to understand the C grammar. The grammar is 948 lines long, which can be called not trivial. It took me for a while to figure out even the starting rule of the grammar. By the way it is compilationUnit, which now seems so obvious, but through the plenty of other rules, it wasn't trivial in the beginning.

To understand the rules I tried to analyse parse trees, but even for simple programs it is big, think about Hello world's parse tree! Sometimes it was hard to find out even where should I zoom into the parse tree, so finding an element took much time.

Another challenge was, that function declaration in C has another form. I didn't know before that this kind of function declaration is valid for C, but it is. So, I had no idea what is the problem with the instrumenter, then I thought this code is not valid, but turned out to be valid. This made me extend the function declaration creator class. It was interesting to find out, how can I differentiate between the different kinds of function declaration and retrieve the name, return type, and arguments and their type.

```
main(argc, argv)
int argc;
char *argv[];
{
  printf("Hello world!\n");
  return 0;
}
```

This grammar can't handle #include-s and macros. Before giving in the code to ANTLR it has to be preprocessed, for this gcc is used, with `-E` flag. Although, it helps dealing with #define directive, it is unconvenient for #include directive, because it copies the whole header, making a 5 lines long hello world program into a 854 lines long source code, and it was just the stdio.h header. So, it can make the code significantly larger. To prevent that, one can delete the headers before preprocessing the code and insert it afterwards. The challenge with it is #include <stdio.h> can have different forms. All definition below is valid:

```
#/* comment here */include/* comment here */<stdio.h>

#include <stdio.h>

#   include          <stdio.h>

#include<stdio.h>
```

Testing my MC/DC transformation wasn't trivial, because of different reasons. First, the instrumenter may not keep the white space character as they were before, because the grammar skips white spaces. This makes the testing hard, because the transformation's output is a string, so the expected output should also be string. So, even a difference in white space character would break the equality of the two strings. To solve this problem I removed every white space character from both strings. Second, *and* and *or* are both commutative, meaning $a \vee b = b \vee a$ and $a \wedge b = b \wedge a$, which can cause a different code syntactically, than expected, but the same semantically. The codes at Figure 5.8a and Figure 5.8b are the same semantically, but different syntactically, so string comparison will say they are not equal. To overcome this difficulty, I examined the output of the transformation and ordered variables to match them, while paying attention not to change the semantics of the code. This testing proved very useful, because I found several bugs, for the first couple of times.

5.3 Backend

Backend is the component that generates inputs for the tests. For this purpose, CBMC is used in the tool. CBMC has a `--cover` flag, which can be set to *assertion*, meaning CBMC tries to generate inputs covering the assert statements inserted at instrumentation phase.

```
if (a == 0) {
  if (b == 0) {
    return 1;
  }
  else {
    return 0;
  }
}
else {
  return 0;
}
```

(a)

```
if (b == 0) {
  if (a == 0) {
    return 1;
  }
  else {
    return 0;
  }
}
else {
  return 0;
}
```

(b)

Figure 5.8: Syntactically different, but semantically equal codes

Model checkers sometimes have problem with loops, so does CBMC. Fortunately, as a bounded model checker it is possible to set its bound, with the `--depth` flag. The question is, what value should be used. It is possible that for a given value, CBMC won't return anything in an acceptable time for one program, but will generate 100% for another program. That means, the execution time cannot be defined by the depth exactly.

5.3.1 Measurement

To get information about the bound, measurements were taken. For this, the codes were taken from SIR (Software-artifact Infrastructure Repository), which is complete infrastructure to support experimentations with testing techniques [10]. Four projects were used, *print_tokens*, *schedule*, *tcas* and *totinfo*.

Bash script was used for measurement, because it would have been cumbersome rewriting the parameters of the function calls each time, moreover this is much faster, because it start the next command immediately after finishing one. In addition for high values, it isn't obvious, when they terminate, maybe in a second, or ten second, or eight minutes, or three hours. A ten minutes timeout was given for each CBMC generation. It is a subjective time limit, but a limit has to be defined. The whole process took thirteen hours, even with this time limit. The bash script can be seen below. It iterates over the functions of the program, and the depth values. So it runs CBMC for each function with each depth, and saves the CBMC's output to a file, and timing data to another file. A little optimisation have been used, if CBMC is timed out for a value, then it obviously will to a greater value, so it should be omitted.

```
FUNCTIONS='initialize ALIM Inhibit_Biased_Climb Non_Crossing_Biased_Climb Non_Crossing_Biased_Descend
      Own_Below_Threat Own_Above_Threat alt_sep_test'
DEPTHS='50 100 150 200 250 300 350 400 450 500 550 600'
cd /to/root/folder/

for FUNCTION in $FUNCTIONS
do
  FILE="${FUNCTION}.c"
  mkdir $FUNCTION
  for DEPTH in $DEPTHS
  do
    { time timeout 10m cbmc $FILE --function $FUNCTION --cover assertion --depth $DEPTH >> "${
     FUNCTION}/result${DEPTH}${FUNCTION}.txt" ; } 2>> "${FUNCTION}/time${DEPTH}${FUNCTION}.txt"
    exit_status=$?
    if [[ $exit_status -eq 124 ]]; then
      echo "Timeout at ${FUNCTION} ${DEPTH}"
      break
    else
      echo "Finished ${FUNCTION} ${DEPTH}"
    fi
```

```
  done
  if [[ $exit_status -eq 124 ]]; then
    { time timeout 10m cbmc $FILE --function $FUNCTION --cover assertion >> "${FUNCTION}/
     resultNODEPTH${FUNCTION}.txt" ; } 2>> "${FUNCTION}/timeNODEPTH${FUNCTION}.txt"
  fi
done
```

The results were organized into Excel documents. Example print_token had the most interesting results, so this is detailed here. It has seventeen functions. This is too much to present on one diagram and some of them behaved the same way, so the interesting ones were chosen, *error_or_eof_case, get_actual_token, get_token, next_state, numeric_case, and print_token.* Figure 5.9 shows the coverage in percent, depending on the depth parameter's value. If a line ends too early, that's because the function timed out for greater depths. Figure 5.10 presents the execution time in seconds depending on the depth parameter's value. The time out limit was ten minutes, which equals to six hundred seconds. In case of next_state function, it exceeded greatly that limit. This anomaly doesn't have any certain explanation. Linux's timeout command was used, maybe CBMC used so much processor time, that it received timeout's kill signal five hundred seconds later.

Looking at the execution results at Figure 5.9, it is not surprising that it is monotonically increasing. Although, there are functions, which timed out, but without time limit, they would have reached higher or the same result as in the earlier execution. Unfortunately, time is a bottleneck in this case, nobody wants to wait days or even more, for some test input. For this reason, a time out limit has to be introduced, ten minutes seemed long enough for generating some inputs, but short enough to be convenient for many executions.

The bad thing about timing aspect, it seems to be exponential in depth. Meaning, for a little increment in the depth causes a huge increase in the execution time. Looking at the execution times at Figure 5.10, three separate types can be observed, these are classified in three categories.

- **Constant:** this category contains functions like print_token. They have nearly constant execution time. It is not constant, they differ in tenths of a second – it cannot be seen on the diagram, because the resolution of its scale is not high enough for this. This category contains functions, independent from the depth parameter.

- **Exponential:** this category contains functions like next_state and get_token. They are surely exponential. Function next_state was executed with 100, as the value of depth, under 35 seconds, while it could not return anything in 1116 seconds, when called with value 150 for depth. This category contains functions, sensitive for the depth parameter. This category restricts the possible values of the depth parameter.

- **Limit-needer:** this category contains functions like get_actual_token, error_or_eof_case and numeric_case. They increase slowly as the depth is getter higher and higher. In case of error_or_eof_case for depth value 50, the execution time was under a second, then it raised linearly to 48 seconds, for depth value 600, but without depth limit, it timed out. It is possible that they are not exponential, only linear. Also it cannot be excluded that above a depth, it start to behave exponentially. To get the exact answer to that question, further measurements would be needed. Although, it is not known if the elements of this category behaves linear or exponential way, it is not examined further, because it does not really matter. Knowing they need a depth limit is enough. This category contains functions, depending on the depth value, but less sensitive, than in the exponential category.

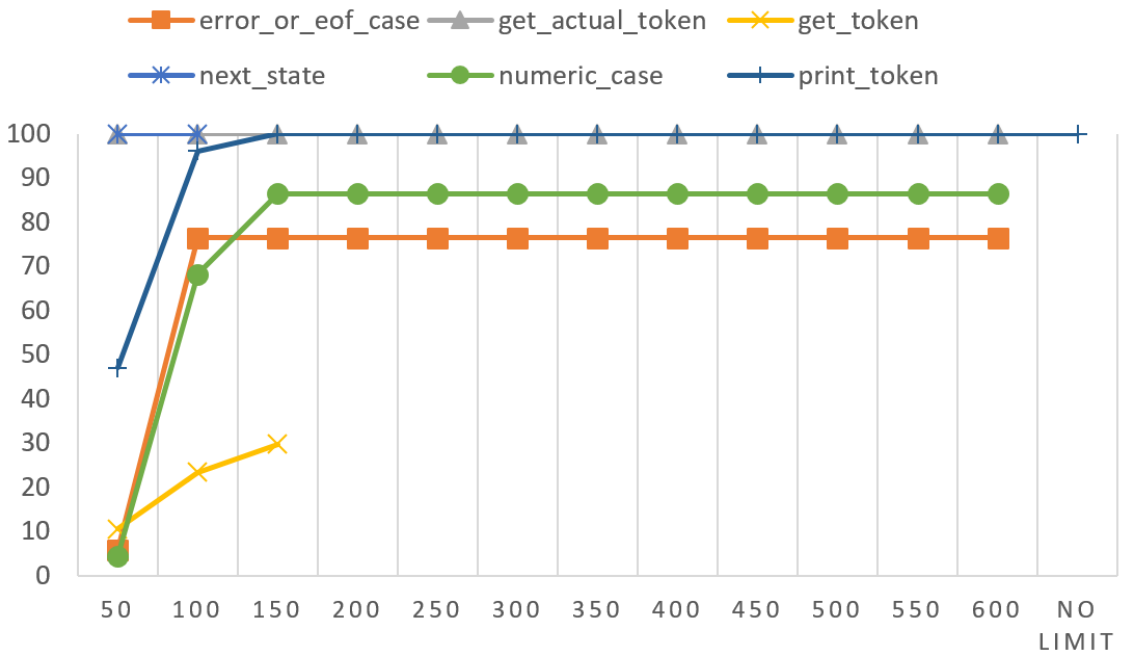


Figure 5.9: CBMC execution results

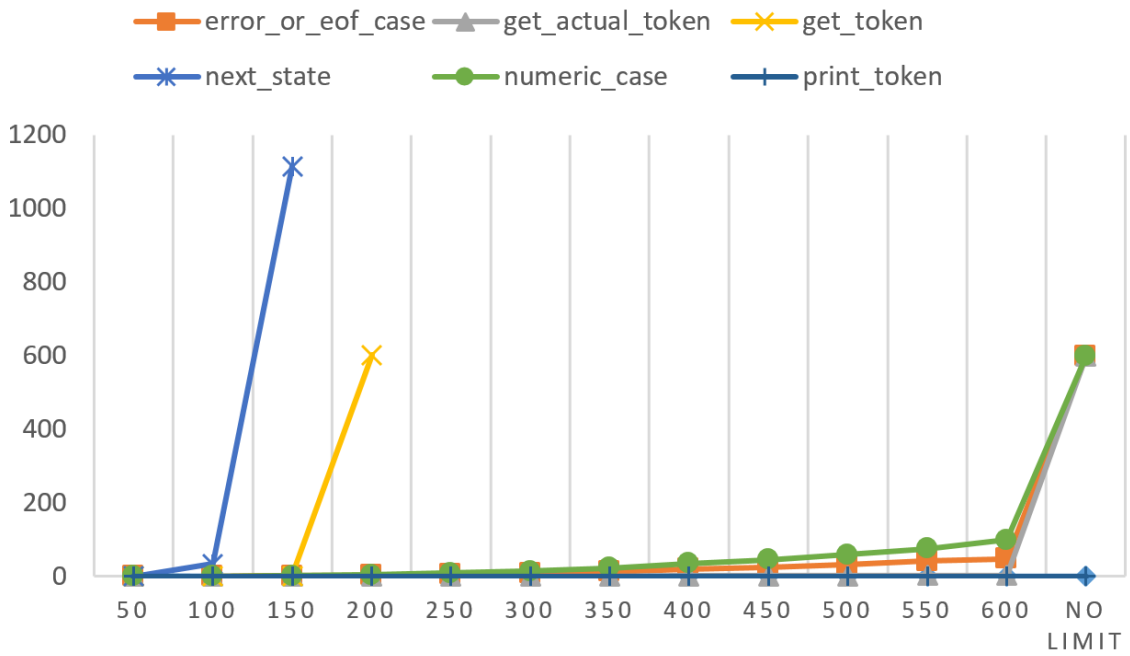


Figure 5.10: CBMC execution time

Let's examine the cause of the different behaviour of the functions! What makes a function belong to exponential category, and an other function to constant category?

The only member of constant category, print_token doesn't have any loop in it, neither calls another function from the code. It calls fprintf, which doesn't really modifies the functioning, and it contains a switch structure, with twenty cases. It seems like switch structure doesn't affect the execution time category. Based on this, it is assumed that neither affects if structure.

It turned out, next_state calls itself, based on this, it can be said that recursion may cause exponential execution time. The other member of exponential category, get_token calls next_state along some more functions. It is quite interesting that next_state times out, for even low depth values, but it reached 100% coverage for the lowest in the measurement.

The members of limit-needer category, except get_actual_token, calls some other functions, while get_actual_token calls only isspace, which is a library function, but it has three for loops inside.

Summarizing the thoughts on this question, it seems recursion greatly affect the execution time, additionally function calls and loops may have effect on the execution time.

Based on the measurement 100 was chosen for depth parameter, because CBMC returned cases for every function, for this parameter and there aren't considerable differences with between 100 and higher values.

Additionally, it may be useful to understand that it may happen that a given assertion is theoretically impossible to cover. In this case, CBMC obviously will not return 100% coverage of the assertions. For example, in case of the code to the right, the assert statement will never be reached, because there is not any number, which is both higher and lower, than another one. Even integer overflow cannot help with this one.

```
if (x > 0) {
  if (x < 0) {
    assert(0 == 1);
    printf("Impossible\n");
  }
}
```

5.3.2 Output parsing

CBMC can create its output in JSON format, which is useful, because it facilitates the parsing. There are off-the-shelf solutions for parsing a JSON document, the tool uses json-simple. Figure 5.11 shows examples for outputs of CBMC in JSON format, Figure 5.11a shows a case, where to integer values were needed by a function, Figure 5.11b shows an example for generating values for a structure, which contains two integer variables.

The task here is to create test variables and organize them into test cases, from JSON files like at Figure 5.11.

It may seem straightforward, but it is not! There are several things that makes this step a complex problem. Here is a list of the difficulties.

Difficulties

- Naming convention is ambiguous. Look at Figure 5.11b! The key "name" occurs in three different contexts. Once it is the name of structures member, twice it is some kind of type of the structure's member. Lastly it is used to denote, that a certain object is a structure, in this case "p" is a structure.

  Fortunately, they cannot be in the same scope, so it is possible to find out, which interpretation is needed. Anyway, it is an unfortunate naming convention, because it may confuse the developer.

- Types are not the same as in the source code. An example code were created, where every possible type was used, then executed with CBMC. The types were not always the same, for example for unsigned long int, CBMC returned "___CPROVER_size_t". To avoid problems coming from using different types, the types are achieved from the function's header, which is created by the instrumenter.

```
"inputs": [
  {
  "id": "x",
    "value": {
      "binary":
     "00000000000000000000000000000000",
      "data": "0",
      "name": "integer",
      "type": "signed int",
      "width": 32
    }
  },
  {
  "id": "y",
    "value": {
      "binary":
     "00000000000000000000000000000001",
      "data": "1",
      "name": "integer",
      "type": "signed int",
      "width": 32
    }
  }
]
```


(a) CBMC output for int variables


```
"inputs": [
  {
  "id": "p",
    "value": {
    "members": [
      {
      "name": "x",
      "value": {
        "binary":
     "00000000000000000000000000000000",
        "data": "0",
        "name": "integer",
        "type": "signed int",
        "width": 32
      }
    },
    {
      "name": "y",
      "value": {
        "binary":
     "00000000000000000000000000000001",
        "data": "1",
        "name": "integer",
        "type": "signed int",
        "width": 32
      }
    }
    ],
    "name": "struct"
    }
  }
]
```


(b) CBMC output for struct with int variables

Figure 5.11: Examples for CBMC output in JSON format

- Temporary variables are used in case of pointers. Usually CBMC uses names for temporary variables, which aren't valid C variable names, they contain $ character. There is an exception. The first time after a pointer variable it uses the name "tmp", which could be a valid variable name, even when other tmp named variable exists in the code. It is ambiguous, which tmp is which. The assumption is, the tmp after a pointer is a temporary variable that contains the value, to which the pointer is pointing.

- Consecutive generations are not separated. If CBMC generates more, than just one test case, it writes the variables one after another, without separating the different test cases. When parsing the JSON file, it is needed to check, if the current variable has been already added to the test case. If not, than it can be added, else a new test case has to be made. Solving this is possible, but cumbersome.

- Structure name is missing. This is the biggest problem. CBMC just writes "struct", where the the name of the structure should be written. It can be solved, when the function parameters contain the structure typed variable, because the instrumenter saves this, and the structure's name can be retrieved by the instance name. The trouble comes, when the structure contains another structure. It is not discovered by the instrumenter, and CBMC does not publish its name, so the type will remain unknown. This limits the set of C programs, to which the tool can generate tests.

5.3.3 Limitations

Since C does not really supports passing arrays between functions, usually a pointer and an integer value is used to pass the array. The pointer points to the first element's address of the array, and the integer value tells the length of the array. Another possibility is "string", which is a null terminated character array. This does not need a length parameter, because with the terminating null character, the end can be determined. The thing is, the developer may know, if only a pointer or a pointer to an array is needed, but CBMC does not know. So, it will not generate values to arrays, because it is not clear that an array is needed. If an array is inside a structure, it is able to generate values for the array.

As mentioned earlier, CBMC does not tell the name of structures, which is a problem when structures are inside another structure, or an array. So, the tool does not support structures inside structures, nor structures inside arrays.

It does not generate temporary variables to pointers in structures. It is a problem, because they would hold the value, to which the pointer is pointing. This way only NULL pointers are put into test cases for pointers inside structures.

5.4 Test execution

At test execution phase, a test code is generated for the chosen C unit test framework, then it is compiled and run. Section 5.4.1 details the first, Section 5.4.2 the latter.

A test case consists of test input, and expected output, which is compared to the actual output, after the test has been executed. The tool does not generate expected output, it has to be given by the tester. This way, the tests are more meaningful, though not fully automated.

5.4.1 Test framework

CUnit has been chosen for test framework, because it seemed simple to use, and its abilities are sufficient.

In the test code, there has to be a main function. This is the entry point of the execution. Inside main function, test functions can be registered. These functions are similar to simple JUnit test functions, but they do not have @Test annotation. CUnit provides several assertion methods, the tool uses CU_ASSERT_EQUAL(actual, expected).

The task here is to create the above mentioned test method, by TestSuite class, which has been created by parsing the output of CBMC.

Figure 5.12 presents the classes used for defining test cases. TestSuite contains a list of TestCases, so they can be managed together. TestSuite is a singleton class, so it may have maximum one instance. At the beginning it has an empty list of test cases, later test cases can be added with the addToTestSuite(TestCase) method, and retrieved with the getTestCases() method.

TestCase stores the test variables in a list, storing TestVariableBase instances. Although, TestVariableBase is abstract, meaning it cannot be instantiated, the instances of its subclasses can be put into the list. TestCase does not return this list, but returns Iterator<TestVariableBase>, letting others iterate over the list. To do this, TestCase has to implement Iterable<TestVariableBase> interface.

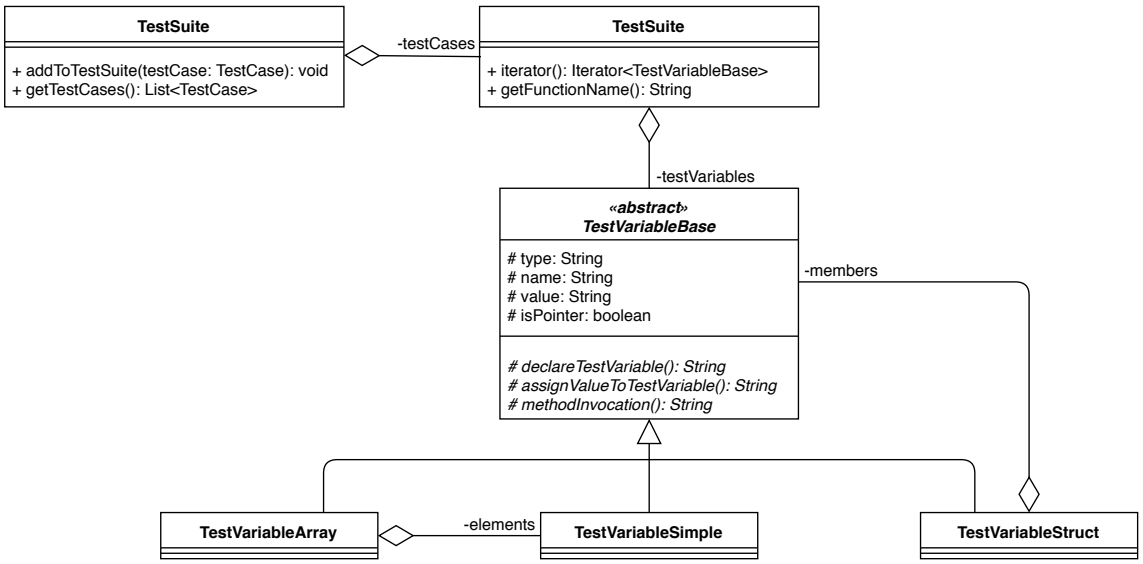


Figure 5.12: Class diagram of the test case classes

TestVariableBase stores the common fields for the different variable types, and defines the possible operations. Each test variable must have a type, name and value, also it has to be known, it the variable is a pointer. Figure 5.13 shows examples for declareTestVariable(), assignValueToTestVariable(), and methodInvocation() methods for the different test variable types; array type can be found at Figure 5.13a, simple type at Figure 5.13b, structure type at Figure 5.13c. In a test case, variables are following each other, and a variable has a declaration, then an assignment. The information, whether a variable is a pointer, is needed first, because the method invocation, second, in case it is a pointer to a structure, then operator -> is needed instead of . operator, to reach its members.

```
// declaration
int array[3];
// assignment
array[0] = 0;
array[1] = 1;
array[2] = 2;
// method invocation
method(array);
```

(a) Array type test variable

```
// declaration
int x;
int* y;
//assignment
x = 0;
y = 1;
// method invocation
method(x, &y);
```

(b) Simple type test variable

```
// declaration
Point point;
// assignment
point.x = 0;
point.y = 1;
point.z = 0;
// method invocation
method(point);
```

(c) Struct type test variable

Figure 5.13: Examples for different test variable types

5.4.2 Compilation

Compilation of C codes are not trivial, for this reason a compilation tool, CMake, is used. This facilitates this phase, as it needs only a textual description, called CMakeLists.txt, which contains informations, such as the location of source files, header files. It is possible to set desired flags in this file used, like it would be used, for example with gcc. The code below shows the content of CMakeList.txt, generated by the tool. The folder structure is fixed, because this way the same CMakeLists.txt can be generated. It looks for the includes (header files) in the include directory, for source files in the test/src directory. It is needed because the main function has to be removed from the code under test, because test code must have a main function, and compilation will fail, if more than one main function is

found. The tool does it automatically, with the help of the instrumenter component. So, the contents of the src folder are copied to test/src to remove the main function, and to avoid generating files into the original folder.

Some flags are added, needed by Gcovr, the evaluator tool. Gcov, cunit, m (math library) are linked to the executable, called test.

```
cmake_minimum_required (VERSION 2.6)
project (Test)

include_directories(include)

file(GLOB SOURCES "test/src/*.c")

add_executable(test ${SOURCES})

find_program(GCOV_PATH gcov)

if(NOT GCOV_PATH)
  message(FATAL_ERROR "gcov not found! :(")
else()
  message("found it")
endif()

add_definitions(-fprofile-arcs -ftest-coverage -fPIC -O0)
target_link_libraries(test gcov cunit m)
```

The commands to the right show the process of the compilation, and execution. With these parameters CMake can be called from arbitrary directory, the path after -B tells where to put its output, -H tells the path to the folder containing CMakeLists.txt. This is useful, because it is executed with Java's Process class, which cannot make use of Linux's cd command. CMake generates a Makefile and other files needed for compilation inside the build folder. After that, make is called, which compiles the program. It is also helpful that make has a parameter -C, for providing the path to the Makefile's folder. If the compilation is successful, an executable file is created inside the build folder, running this will run the tests.

```
# Run CMake
cmake -B/path/to/build/ -H/path/to/root/
# Compile with make
make -C /path/to/build
# Run the executable
/path/to/build/test
```

5.5 Test evaluation

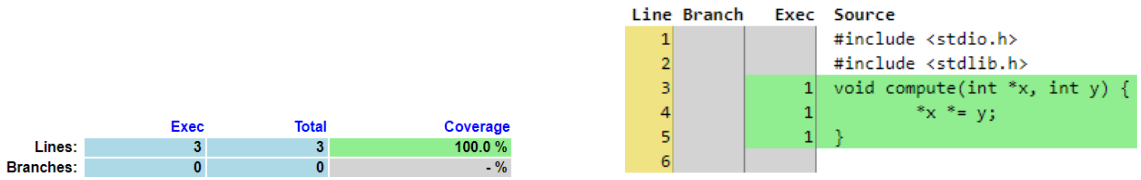
The aim of this phase is to give the tester a visual feedback on the reached coverages. Line colouring can help the understanding, as the colouring the reached lines to green and others to red is very expressive. Moreover a table is used, to quantify the results.

Gcovr is used in this phase, it generates html report on the line and branch coverage. It can be run with the following command from command line. The tool runs this automatically, except for test generation or test execution use cases.

```
gcovr -r /path/to/rootfolder --html --html-details -o /path/to/output/evaluation.html
```

Figure 5.14 shows the table of metrics, and the lines, 5.14a presents the table of metrics, Figure 5.14b the coloured lines.

There is a thing that could be better in Gcovr, if branch coverage is not 100%, it does not tell which branch is not covered. Figure 5.15 shows that branch coverage is 75%, Figure 5.16 shows the line, which has and uncovered branch. But it cannot be defined by these informations, which branch has not been covered.



	Exec	Total	Coverage
Lines:	3	3	100.0 %
Branches:	0	0	- %

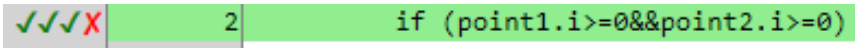
Line	Branch	Exec	Source
1			#include <stdio.h>
2			#include <stdlib.h>
3		1	void compute(int *x, int y) {
4		1	*x *= y;
5		1	}
6			

(a) Table of coverage results **(b)** Coloured source code

Figure 5.14: Gcovr generated evaluation

Branches:	3	4	75.0 %

Figure 5.15: 75% branch coverage



✓✓✓✗	2	if (point1.i>=0&&point2.i>=0)

Figure 5.16: Uncovered branch

Chapter 6

Evaluation

This chapter evaluates the tool. In Section 6.1 unit tests are introduced that were used to test the certain units of the tool. Section 6.2 shows results of the tool's execution for simple programs, while Section 6.3 does the same, besides using real world problem, instead of simple programs. Last section, Section 6.4 the results are summarized, a real picture is given from the tool's capabilities.

6.1 Unit tests

Some of the most important functionalities of the tool have been tested with unit tests, these are the MC/DC transformation, chosing the right test strategy based on command line arguments, and converting test variables into C code. In the following paragraphs these are detailed, the test cases are organized into tables, and the result of the unit test execution are inserted as figures.

MC/DC transformation tests The implementation of this feature was not trivial, it was a typical must test with unit test thing. In the implementation chapter it has been mentioned, that it was hard to test this, because of the comparison of strings. During development, these test pointed out several flaws of the transformation. It was funny, that it worked for complex expressions, but then a simple case without logical operator was introduced, and it made the tests fail. In Table 6.1 the inputs of the test cases are collected, the Name column shows the name of the given test case, the Expression column shows the input expression that had to be transformed. Outputs are not provided here, because they are so long. The aim was to cover every possible combinations of AND and OR and Negation.

There are 33 test cases. Figure 6.1 shows that all passed successfully. Based on this, MC/DC transformation seems to be correct.

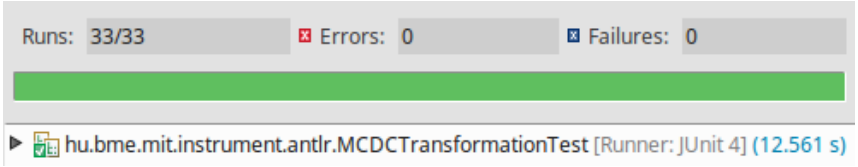


Figure 6.1: MC/DC transformation test results

Name	Expression
ifAfterIf	if (b > 0) {...} else if (c < 1) {...}
ifInsideIf	if (b > 0) { if (c < 1) {...} }
negatedAnd	if(!(a == 1 && b == 1))
negatedNegatedOneCondition	if (!(!(a == 1)))
negatedOneCondition	if (!(a == 1))
negatedOr	if (!(a == 1 || b == 1))
oneCondition	if (a == 1)
simpleAnd	if (a == 1 && b == 1)
simpleOr	if (a == 1 || b == 1)
ifInsideSwitch	switch(a){case 1:if(b>0||c<1){...} case 2:if(b<=0&&c>=1){...}}
outsideSwitch	switch(a){case 1:... case 2:... default:...} if (b > 0 || c < 1)
simpleSwitch	switch(a){case 1:... case 2:... default:...}
switchInsideIfWithAnd	if (b > 0 && c < 1) { simpleSwitch }
switchInsideIfWithOr	if (b > 0 || c < 1) { simpleSwitch }
switchInsideSwitch	switch(a) { case 1: switch(z) { case 9:... case 8:...}...}
neg_neg_a_And_neg_b	if (!(!(a == 1) && !(b == 1)))
neg_neg_a_Or_neg_b	if (!(!(a == 1) || !(b == 1)))
a_AND_b_Or_c	if (a == 1 && b == 1 || c == 1)
a_And_b_And_c	if (a == 1 && b == 1 && c == 1)
a_And_b_OR_a_And_c	if ((a == 1 && b == 1) || (a == 1 && c == 1))
a_And_b_OR_c	if (a == 1 && (b == 1 || c == 1))
a_OR_b_And_c	if ((a == 1 || b == 1) && c == 1)
a_Or_b_AND_c	if (a == 1 || b == 1 && c == 1)
a_Or_b_Or_c	if (a == 1 || b == 1 || c == 1)
sevenVariables	((((a) && !(b)) || (!(c) || (d))) && !(!((e) || !(f)) && (g)))
neg_a_AND_b_Or_c	if (!(a == 1 && b == 1 || c == 1))
neg_a_And_b_And_c	if (!(a == 1 && b == 1 && c == 1))
neg_a_And_b_OR_a_And_c	if (!((a == 1 && b == 1) || (a == 1 && c == 1)))
neg_a_And_b_OR_c	if (!(a == 1 && (b == 1 || c == 1)))
neg_a_OR_b_And_c	if (!((a == 1 || b == 1) && c == 1))
neg_a_Or_b_AND_c	if (!(a == 1 || b == 1 && c == 1))
neg_a_Or_b_Or_c	if (!(a == 1 || b == 1 || c == 1))
neg_sevenVariables	(!((((a) && !(b)) || (!(c) || (d))) && !(!((e) || !(f)) && (g))))

Table 6.1: Test cases for MC/DC transformation

Test strategy creator test The aim of this test, is to ascertain about the correct functionality of ArgumentParser, which decides about the test strategy to use, based on the command line arguments. If there is a problem with the arguments then DoNothingStrategy has to be returned. If user chose generation (-gen or --generate), then GenerationStrategy, else if execution (-exe or --execute), then TestExecutionStrategy, else if evaluation (-eval or --evaluate), then EvaluationStrategy, else EveryStepStrategy.

Table 6.2 shows inputs, command line arguments, and the expected test strategy.

Figure 6.2 shows all the nine test was successful. The ArgumentParser decides well about the desired test strategy.

Input	Expected strategy
"-r /home/meres/ –input struct.c -mcdc"	EveryStepStrategy
""	DoNothingStrategy
"-r /home/ –input something.c –generate"	GenerationStrategy
"–generate"	DoNothingStrategy
"-r /home/ -in example.c -exe"	TestExecutionStrategy
"-r /home/ -exe"	DoNothingStrategy
"-r /home/ -eval"	DoNothingStrategy
"-r /home/ -eval -in example.c"	EvaluationStrategy
"-r /home/ -eval -in example.c -nosuchoption"	DoNothingStrategy

Table 6.2: Test strategy creator test cases

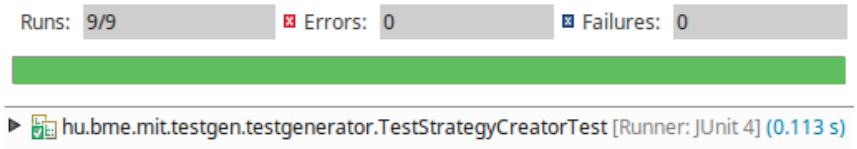


Figure 6.2: Test strategy creator test results

Test variables test It is important to be sure that each subclass of TestVariableBase returns the proper string for the declaration, assignment, and method invocation of that variable. Table 6.3 contains the test cases. The aim was to cover all default possibility of variables, such as arrays, structures, and simple variables. In case of compound variables, the elements are listed below, and the assignment is defined for them. Columns Declaration, Assignment, and Method contains the expected values.

Type	Input	Declaration	Assignment	Method
simple	int i 0	int i;	i = 0;	i
simple	char c 65	char c;	c = 65;	c
simple	int* i 0 true	int* i;	i = 0;	&i
struct	Point point	Point point;		point
simple	double x 0.0		point.x = 0.0	
simple	double y 1.0		point.y = 1.0	
struct	Point point	Point point;		point
simple	double* x 0.0 true		point.x = 0.0	
simple	double* y 1.0 true		point.y = 1.0	
struct	P p true	P p =(P)malloc(sizeof(struct P));		p
simple	double x 0.0		p->x = 0.0	
simple	double y 1.0		p->y = 1.0	
array	int array	int array[3];		array
simple	int x 0		array[0] = 0;	
simple	int y 0		array[1] = 1;	
simple	int z 0		array[2] = 2;	

Table 6.3: Test variables test cases

Figure 6.3 shows all 21 test case was successful. Table 6.3 contains seven lines, so it is not trivial, how it became 21. For each test variable, the declaration, the assignment, and the

method invocation are tested separately, so three tests are run for seven variables, which turns out to be twenty one.

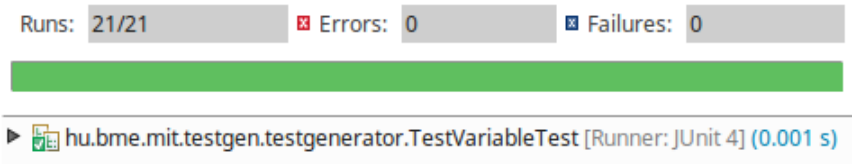


Figure 6.3: Test variables test results

6.2 Simple programs under test

Before using the tool on programs from the real world, it makes sense to examine its behaviour on simple elements of the language. The tool was executed on short programs, with only one trait, for example, the function gets pointer, or structure, or calls itself, etc.

Table 6.4 shows these different simple building blocks that has been tested separately. **I** stands for instrumentation, its value is **y**, if the instrumentation was successful, **n** otherwise. **B** stands for backend, its value is **y**, if the test input generation was successful, **n** otherwise. **Ex** stands for execution, its value is **y**, if the program was compiled and executed successfully, **n** otherwise. **Ev** stands for evaluation, its value is **y**, if the html output generation was successful, **n** otherwise. **Error** describes the problem with the given program, if it is empty, then the whole process was completed successfully.

For all program containing structure, a header file has to be created. Otherwise the test code cannot be compiled, because it does not know that structure. It is a good practice to define the type of the structure to the same name, as the structure's name. The tool needs this, when memory allocation is needed for the structure, because sizeof() needs the struct's name, which is not saved in the instrumentation phase.

The tool can handle most of the elements of C, but it has limits. For example, it cannot handle structures inside structures, or multidimensional arrays.

6.3 Real world programs under test

Real world programs are the four program from SIR, which has already been introduced due to CBMC measurement, in Section 5.3.1.

print_tokens Instrumentation runs successfully. But problem occurs during running the backend. The problem is a bit complex. To run instrumentation successfully, the #includes are deleted from the code, and saved in an other file. After that preprocessing is done, which is followed by the instrumentation. Then #includes are inserted into the beginning of the instrumented C code. The problem there, is a typedef in the code, which makes BOOLEAN a valid type. In the original source code, an include can be found after that typedef, so in the header, BOOLEAN type can be used. Unfortunately, during instrumentation phase the information about the #include's line is lost, so the tool does not know if it has to be put in a dedicated line. So, it puts in the first lines. In this case it causes an error, because the header uses a type which is not defined that time. For this reason CBMC cannot generate inputs.

Name	I	B	Ex	Ev	Error
double_pointer	y	y	n	n	Test cannot be run, segmentation fault.
for_test	y	y	y	y	
function_call	y	y	y	y	
global_variable	y	y	y	y	
if_test	y	y	y	y	
matrix	y	n	n	n	The tool does not support array inside array.
more_testcase	y	y	y	y	
motivation_example	y	y	y	y	
pointer	y	y	y	y	
pointer_more	y	y	y	y	
pointer_tmp	y	y	y	y	
recursion_test	y	y	n	n	Input is too large for factorial
simple	y	y	y	y	
struct_array	y	y	y	y	Fails without header file.
struct_array_struct	y	n	n	n	The tool does not support struct inside array.
struct_inside_struct	y	n	n	n	The tool does not support struct inside struct.
struct_is_pointer	y	y	y	n	If struct name equals typedef name, then works.
struct_simple	y	y	y	y	
struct_with_pointer	y	y	n	n	Test cannot be run, segmentation fault.
struct_with_pointer_to_array	y	y	n	n	Test cannot be run, segmentation fault.
switch_test	y	y	y	y	
while_test	y	y	y	y	

Table 6.4: Evaluation of simple programs

There may be more solutions for this problem. First, if the grammar could handle #include-s and #define-s, then this problem would not occur. Second, the tool could handle somehow this case. It is a bit complex, to find where the #include lines should be complex, because the line numbering may change during preprocessing.

schedule This program consists only of a single C file, schedule.c. If the tools is run only on that, then it cannot build the code, because the test code needs a definition of those structures.

If a header file was created manually, that contained the structures, then the it could be compiled and run. Unfortunately, the test execution fails at some point, due to segmentation fault. It happens when a parameterless void function is called. This function tries to access a global variable, which was not initialized, this may cause the segmentation fault. The tool is not able, to create header files automatically, it is the tester's responsibility to provide it for the tool.

This case shows deficiencies of the tool in pointer and global variable handling.

tcas The tool successfully generated tests for this program, although its coverage is not the best. This may be due to CBMC cannot generate 100% coverage for some functions. Figure 6.4 shows the coverage results.

	Exec	Total	Coverage
Lines:	31	42	73.8 %
Branches:	8	64	12.5 %

Figure 6.4: Coverage result of tcas

totinfo The tool was running for unexpectedly long time for this code, almost for an hour. It could instrument the code, generate inputs, but the test code became somewhat faulty. A double variable should be in the test case, but instead a null stands for the type. After chasing the bug for a while, it seems like the tool does not interprets *double a, x;* well. The type of a is made null, while x stays double. It must be a bug in the function definition creation.

This bug comes from the instrumenter component. It expects variable declarations in separate statements.

6.4 Summary

As it was expected the tool does not beat the existing C test generation tools. The tool has some limitation, and flaws. This is not surprising, because C projects are not robust, meaning the lack of a header file, or an #include at the wrong position may deny the compilation of the code. Also the different possibilities harden the life of such a tool, for example the function definition can be done more ways.

The tool supports several elements of the C language, but not all of them. It is able to handle simple types, such as int, char, long, float, double; compound types, like structures, arrays; pointers. Then again it has problem dealing with #include-s, #define-s, global variables.

All in all, I think the tool satisfies the tasks written in the thesis task description.

Chapter 7

Conclusion

In this chapter I conclude my thesis work, show possibilities to improve and extend the tool, and express my experiences that I gained during working on this tool and thesis.

7.1 Summary of results

During the thesis, the first thing I did, was doing researches on the area of testing, and test generation. Then I tried existing test generation tools for C language. Based on the knowledge, experiences, and observations of the afore mentioned activities, I designed the tool's architecture, defined what kind of dependencies are going to be needed. Then I looked for a tool for every phase and tried them. After the tools were proved to work, I started the implementation. Moving from step by step, the tool worked for more and more complex codes. Unfortunately, it has several limitations. This may be the cause of the characteristic of the used tools, but all in all I think it is, because the complexity of C programs. My opinion is it is not hopeless to upgrade this tool to be useful, but it requires a vast amount of effort. The possibilities for improvement are detailed in the next section.

To summarize my work, I showed the general methods of software verification, mostly methods for testing, with particular regard to different ways of test generation. I designed a tool, which is capable of generating test to programs written in C language. I implemented the tool, and examined and evaluated it through own, and real world C codes.

7.2 Future works

There are several possibilities to improve the tool. Some of the most crucial ones are detailed below.

When I developed the tool, I usually know what was the cause of a problem, if the tool did not work. During evaluation I noted, the tool does not write useful error messages to the user. It is not because I did a lazy job on that, maybe could have written some more messages, but because determining if a used tool terminated successfully is not easy. For example, if CBMC cannot generate test due to errors in the code, it prints message to the console and returns normally, the same goes for CMake and make. It is hard, to parse those kind of messages. Maybe the tool should print them, and that would help the user to recognize the problem.

The tool does not support structures/arrays inside structures/arrays. In the instrumentation phase the structure could be saved somewhere, making it possible to determine the names of structures inside structures, when test cases are translated from CBMC's JSON output. This could be useful for generation of header files as well.

Although, default value is given for expected output, the tester should specify the expected output for a given input. An other approach is running the test and use the observed return values, and use them as expected output. To be honest, I do not think this is very useful, because this way we do not gain information about the code's relationship to the specification. I think the tester should be happy to get inputs, and see what output does (s)he expect for that input, and what is the actual output. On the other hand, these automatically generated test outputs may be used for regression testing, if it has been proven that the code works well.

Reaching a given line can be a desired use case, and may be implemented without much effort.

Static analysis could be used to predicate the bound for CBMC.

Random algorithm could be introduced along CBMC.

It would be nice if we could know which branch was not covered, by the evaluation report. Maybe it can achieved, by running the test and evaluation for the MC/DC transformed code instead of the original source code.

The architecture of the tool is flexible, so other tools can be introduced at different phases. I have already taken a look at the possibilities. Although, I did not try them, but they seemed usable, by their description.

- Compiler: clang.
- Model-checker: BLAST, CPAchecker, divine4, ESBMC, LLBMC, SATABS, Theta.
- Test framework: CuTest, AceUnit, MinUnit, Seaetest, tinytest.
- Test evaluation: COVTOOL.

I think there are so many possibilities that another thesis could be written about upgrading the tool.

7.3 Subjective experiences

I think I learned a lot, while doing my thesis work. My knowledge of testing, and test generation is expanded. Also, my practical skills improved, thank to installing different tools on Linux and look for the libraries, binaries; using Docker; compile code manually, using gcc. These are very important things, but they are not really taught at the university.

Designing the architecture of this tool was quite enjoyable for me, it was motivating to think about it should be extended with new tools easily. Though, I thought I know C, it turned, C can always show me something new.

I also really liked ANTLR and parsing C codes. That was great, when the MC/DC transformation worked for the first time. It was very challenging, which made it so good, when it worked.

To sum up my thought about this thesis, I am glad that I have chosen this.

Bibliography

[1] What is V-model advantages, disadvantages and when to use it? http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/.

[2] V-model (software development). https://en.wikipedia.org/wiki/V-Model_(software_development), 2016.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Greg Tobin, 2 edition, 2007. ISBN 200602433.

[4] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Software*, 86(8):1978 – 2001, 2013. DOI: 10.1016/j.jss.2013.02.061.

[5] Andrea Arcuri and Lionel Briand. Adaptive random testing: An illusion of effectiveness? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 265–275, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0562-4. DOI: 10.1145/2001420.2001452. URL http://doi.acm.org/10.1145/2001420.2001452.

[6] Suhabe Bugrara and Dawson Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 199–212, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2535461.2535486.

[7] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-123.html.

[8] T. Y. Chen, R. Merkel, P. K. Wong, and G. Eddy. Adaptive random testing through dynamic partitioning. In *Fourth International Conference onQuality Software, 2004. QSIC 2004. Proceedings.*, pages 79–86, Sept 2004. DOI: 10.1109/QSIC.2004.1357947.

[9] Edmund Clarke and Daniel Kroening. Ansi-c bounded model checker user manual. Technical report, 2006.

[10] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, Oct 2005. ISSN 1573-7616. DOI: 10.1007/s10664-005-3861-2. URL https://doi.org/10.1007/s10664-005-3861-2.

[11] Exida. IEC 61508 Overview Report, 2006.

[12] Patrice Godefroid. Test Generation Using Symbolic Execution. In *Annual Conf. on FSTTCS*, pages 24–33, 2012. DOI: 10.4230/LIPIcs.FSTTCS.2012.24.

[13] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite. In *ASE*, 2010.

[14] Dr. Majzik István. Software and System Verification, 2017. Dependability analysis.

[15] Dr. Majzik István and Dr. Micskei Zoltán. Software and System Verification, 2017. Overview of Verification and Validation techniques.

[16] J. C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 547–550, May 2002.

[17] Andrew Kornecki and Janusz Zalewski. Certification of software for real-time safety-critical systems: state of the art. *Innovations in Systems and Software Engineering*, 5(2):149–161, Jun 2009. ISSN 1614-5054. DOI: 10.1007/s11334-009-0088-1. URL https://doi.org/10.1007/s11334-009-0088-1.

[18] Stefan Krisoa, Christopher Temple, Berthold Arends, Pierre Metz, and Bernd Enser. Functional Safety in accordance with ISO 26262, 06 2012.

[19] Virgile Prevosto. ACSL Mini-Tutorial.

[20] Rail Safety and Standards Board. Guidance on high-integrity software-based systems for railway applications, 03 2017.

[21] MISRA The Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Critical Systems*. 03 2012. ISBN 9781906400101.

[22] Dr. László Zoltán. Software technology, 2014. Software process.

Appendix

A.1 Content of the supplement

A.1.1 Prerequisite

The tool runs only on Linux operating system.

1. Java Runtime: `https://www.java.com/en/download/`
2. gcc: `https://gcc.gnu.org/install/index.html`
3. CBMC: `https://www.cprover.org/cbmc/`
4. CUnit: `https://sourceforge.net/projects/cunit/`
5. CMake: `https://cmake.org/download/`
6. Gcovr: `https://gcovr.com/installation.html`

A.1.2 How to run the tool

Use java -jar ctestgentool.jar, -r specifies the root directory of your project, -in specifies the name of the C file under test. To get more help use -h, or --help!

```
java -jar ctestgentool.jar -r /path/to/root/folder/of/your/project/ -in code\_you\_want\_to\_test.c
```

A.1.3 Folder structure

Evaluation Evaluation folder contains C codes, to which the tool has been executed. They also contain the result of execution.

Simple Evaluation/simple contains simple C programs.

Realworld Evaluation/realword contains the names of programs executed. I do not have authority to share the codes, provided by SIR. The codes can be download from here: https://sir.csc.ncsu.edu/portal/index.php

src Src folder contains the source code of the tool.

A.1.4 ctestgentool.jar

This is the executable file of the tool. See section *How to run the tool* for information on how to run the tool!

A.2 Implementation phases on the motivation example

The test generation was run with -mcdc flag.

A.2.1 Output of instrumentation

```
#include <assert.h>

int isDangerousSituation(int boundaryTemperature, int temperature1, int temperature2, int
     temperature3) {
  if (temperature1>boundaryTemperature)  {
    if (temperature2>boundaryTemperature)  {
      assert(0 == 1);
      return 1;
    }
    else if (temperature3>boundaryTemperature)   {
      assert(0 == 2);
      return 1;
    }
    else  {
      assert(0 == 3);
      return 0;
    }
  }
  else if (temperature2>boundaryTemperature)   {
    if (temperature3>boundaryTemperature)  {
      assert(0 == 4);
      return 1;
    }
    else  {
      assert(0 == 5);
      return 0;
    }
  }
  else  {
    assert(0 == 6);
    return 0;
  }
}
```

A.2.2 Output of CBMC

```
"inputs": [
{
  "id": "boundaryTemperature",
  "value": {
    "binary": "00000000000000000000000000000010",
    "data": "2",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "00000000000000000000000000000011",
    "data": "3",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "00000000000000000000000000000011",
```

```
    "data": "3",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "00000000000000000000000000000011",
    "data": "3",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "boundaryTemperature",
  "value": {
    "binary": "00000000000001000000000000000010",
    "data": "131074",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "00000000000000000000000000000011",
    "data": "3",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "00000000000001000000000000000011",
    "data": "131075",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "00000000000001000000000000000011",
    "data": "131075",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "boundaryTemperature",
  "value": {
    "binary": "00000000000001000000000000000010",
    "data": "131074",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "00000000000000000000000000000011",
    "data": "3",
    "name": "integer",
```

```
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "10000000000000010000000000000011",
    "data": "-2147352573",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "00000000000000010000000000000011",
    "data": "131075",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "boundaryTemperature",
  "value": {
    "binary": "10100000000000010000000000000010",
    "data": "-1610481662",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "10100000000000000000000000000011",
    "data": "-1610612733",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "00100000000000010000000000000011",
    "data": "537001987",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "10000000000000010000000000000011",
    "data": "-2147352573",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "boundaryTemperature",
  "value": {
  "binary": "00100000000000010000000000000010",
    "data": "537001986",
    "name": "integer",
    "type": "signed int",
    "width": 32
```

```
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "00101000000000000000000000000011",
    "data": "671088643",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "10100000000000100000000000000011",
    "data": "-1610481661",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "01000000000000100000000000000011",
    "data": "1073872899",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "boundaryTemperature",
  "value": {
    "binary": "01100000000000100000000000000010",
    "data": "1610743810",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature1",
  "value": {
    "binary": "01101000000000000000000000000011",
    "data": "1744830467",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature2",
  "value": {
    "binary": "11100000000000100000000000000011",
    "data": "-536739837",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
},
{
  "id": "temperature3",
  "value": {
    "binary": "00000000000000100000000000000011",
    "data": "131075",
    "name": "integer",
    "type": "signed int",
    "width": 32
  }
}
```

```
]
```

A.2.3 Test code

```
#include <CUnit/CUnit.h>
#include <CUnit/Basic.h>

void test1(void) {
  int boundaryTemperature;
  boundaryTemperature = 2;

  int temperature1;
  temperature1 = 3;

  int temperature2;
  temperature2 = 3;

  int temperature3;
  temperature3 = 3;

  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 1);
}

void test2(void) {
  int boundaryTemperature;
  boundaryTemperature = 131074;

  int temperature1;
  temperature1 = 3;

  int temperature2;
  temperature2 = 131075;

  int temperature3;
  temperature3 = 131075;

  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 1);
}

void test3(void) {
  int boundaryTemperature;
  boundaryTemperature = 131074;

  int temperature1;
  temperature1 = 3;

  int temperature2;
  temperature2 = -2147352573;

  int temperature3;
  temperature3 = 131075;

  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 0);
}

void test4(void) {
  int boundaryTemperature;
  boundaryTemperature = -1610481662;

  int temperature1;
  temperature1 = -1610612733;

  int temperature2;
  temperature2 = 537001987;

  int temperature3;
  temperature3 = -2147352573;
```

```
  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 0);
}

void test5(void) {
  int boundaryTemperature;
  boundaryTemperature = 537001986;

  int temperature1;
  temperature1 = 671088643;

  int temperature2;
  temperature2 = -1610481661;

  int temperature3;
  temperature3 = 1073872899;

  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 1);
}

void test6(void) {
  int boundaryTemperature;
  boundaryTemperature = 1610743810;

  int temperature1;
  temperature1 = 1744830467;

  int temperature2;
  temperature2 = -536739837;

  int temperature3;
  temperature3 = 131075;

  CU_ASSERT_EQUAL(isDangerousSituation(boundaryTemperature, temperature1, temperature2, temperature3)
     , 0);
}


int main() {
  CU_pSuite pSuite = NULL;

  if (CUE_SUCCESS != CU_initialize_registry())
    return CU_get_error();

  pSuite = CU_add_suite("Suite_1", NULL, NULL);
  if (NULL == pSuite) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  if ((NULL == CU_add_test(pSuite, "test1", test1))) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  if ((NULL == CU_add_test(pSuite, "test2", test2))) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  if ((NULL == CU_add_test(pSuite, "test3", test3))) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  if ((NULL == CU_add_test(pSuite, "test4", test4))) {
    CU_cleanup_registry();
    return CU_get_error();
  }
```

```
  if ((NULL == CU_add_test(pSuite, "test5", test5))) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  if ((NULL == CU_add_test(pSuite, "test6", test6))) {
    CU_cleanup_registry();
    return CU_get_error();
  }

  CU_basic_set_mode(CU_BRM_VERBOSE);
  CU_basic_run_tests();
  CU_cleanup_registry();
  return CU_get_error();
}
```

The generated test inputs have been examined, and the expected output has been set accordingly, manually.

```
    CUnit - A unit testing framework for C - Version 2.1-3
    http://cunit.sourceforge.net/


Suite: Suite_1
  Test: test1 ...passed
  Test: test2 ...passed
  Test: test3 ...passed
  Test: test4 ...passed
  Test: test5 ...passed
  Test: test6 ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
              tests       6      6      6      0        0
              asserts     6      6      6      0      n/a

Elapsed time =    0.000 seconds
```

A.2.4 Coverage report

GCC Code Coverage Report

Directory:	testgen/src/		Exec	Total	Coverage
File:	testgen/src/motivation_example.c	Lines:	8	8	100.0 %
Date:	2018-12-10 02:14:30	Branches:	10	10	100.0 %

Line	Branch	Exec	Source
1		6	int isDangerousSituation(int boundaryTemperature, int temperature1, int temperature2, int temperature3) {
2	✓✓	6	if (temperature1>boundaryTemperature) {
3	✓✓✓✓	3	if (temperature2>boundaryTemperature||temperature3>boundaryTemperature) {
4			
5		2	return 1;
6			}
7			else {
8			
9		1	return 0;
10			}
11			}
12	✓✓✓✓	3	else if (temperature2>boundaryTemperature&&temperature3>boundaryTemperature) {
13			
14		1	return 1;
15			}
16			else {
17			
18		2	return 0;
19			}
20			}
21			

Generated by: GCOVR (Version 4.1)

Figure A.2.1: Result of Gcovr evaluation