VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TESTING OF GENERATED C COMPILERS FOR PROCESSORS IN EMBEDDED SYSTEMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. LUDĚK DOLÍHAL
AUTHOR

BRNO 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

TESTOVÁNÍ GENEROVANÝCH PŘEKLADAČŮ JAZYKA C PRO PROCESORY VE VESTAVĚNÝCH SYSTÉMECH

TESTING OF GENERATED C COMPILERS FOR PROCESSORS IN EMBEDDED SYSTEMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR
Ing. LUDĚK DOLÍHAL

VEDOUCÍ PRÁCE
SUPERVISOR
prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2016

Abstrakt

Vestavěné systémy se staly nepostradatelnými pro náš každodenní život. Jsou to obvykle úzce zaměřená, vysoce optimalizovaná, jednoúčelová zařízení. Jádro vestavěných zařízení obvykle tvoří jeden nebo více aplikačně specifických instrukčních procesorů. Tato disertační práce se zaměřuje na problematiku testování nástrojú pro návrh aplikačně specifických procesorů a následně i samotných aplikačne specifických procesorů. Snahou bylo vytvořit systém, ve kterém bude možné otestovat jednotlivé nástroje, jako například překladač, assembler, disassembler, debugger. Nicméně vyvstává také potřeba provádět složitější testy, například integrační, které zaručí, že mezi jednotlivými nástroji nevzniká nekompatibilita. Autor vytvořil s podporou průběžně integračního serveru prostředí, které napomáhá odhalování a odstraňování chyb při návrhu aplikačně specifických procesorů a které je navíc do značné míry automatizované.

Abstract

Embedded systems have become essential for our everyday lives. They are usually highly specialized and optimized single purpose devices. The cores of these devices are usually composed of one or more application specific instruction-set processors. This dissertation thesis focuses on testing of tools for design of application specific instruction set processors (ASIP) and ASIPs themselves. The aim is to create a system which allows testing of tools, such as a compiler, an assembler, a disassembler or a debugger. Nevertheless, there is also need for more complex tests, for example, integration tests which ensure there is no incompatibility between the tools. The author created, with the support of a continuous integration server, an environment that helps to reveal and fix errors during the design of application specific processors and, moreover, this environment is automatized up to a certain point.

Klíčová slova

Testování, překladače, průběžná integrace, hardware software codesign, procesory s aplikačně specifickou instrukční sadou, jazyky pro popis architektury, vestavěné systémy.

Keywords

Testing, compilers, continuous integration, hardware software codesign, application specific instruction set processors, architecture description languages, embedded systems.

Citace

Luděk Dolíhal: Testing of generated C compilers for processors in embedded systems, disertační práce, Brno, FIT VUT v Brně, 2016

Testing of generated C compilers for processors in embedded systems

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením prof. Ing. Tomáše Hrušky, CSc.

. . . .
Luděk Dolíhal
November 23, 2016

Poděkování

Na tomto místě bych rád poděkoval svému školiteli profesoru Tomáši Hruškovi za jeho vedení, čas, rady a velkou podporu, kterou mi poskytoval během mého studia. Dále bych rád poděkoval kolegům, především Karlu Masaříkovi, Zdeňku Přikrylovi, Adamu Husárovi, Ondřeji Ilčíkovi, Liboru Vašíčkovi, Róbertu Baručákovi, Filipovi Matiovskému, Milanu Skálovi a dalším členům Codasip týmu za jejich skvělou spolupráci a nápady.
V neposlední řadě také velmi děkuji svým rodičům Františkovi a Janě Dolíhalovým a své přítelkyni Juliáně Krejčové bez nichž by tato práce nikdy nemohla vzniknout.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

This thesis is going to deal with the area of hardware software codesign and will mainly focus on testing and stability of such tools. Every piece of software contains errors and tools for hardware software codesign are not an exception. It is a well-known fact that the later the error is discovered in the software, the more expensive the process of fixing it is. This fact is shown in Fig. 1.1.

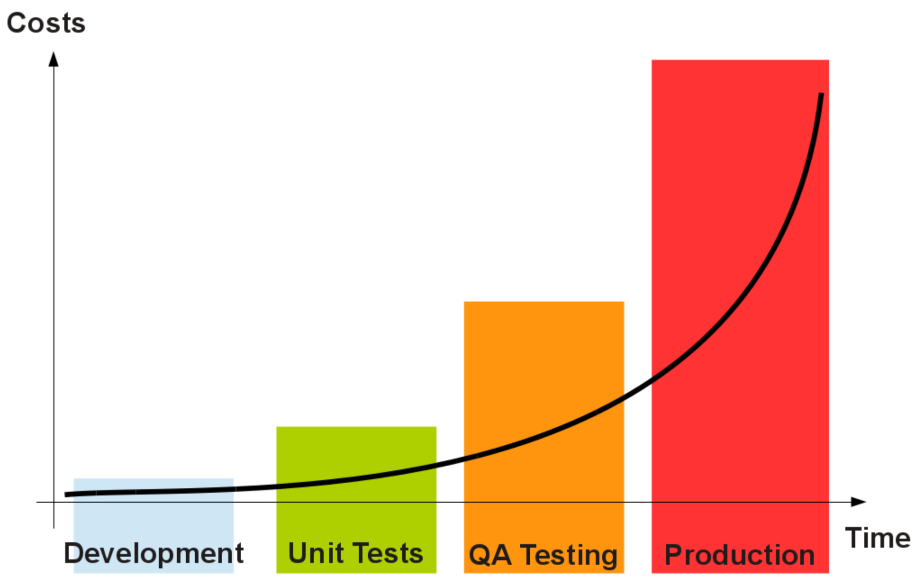

Figure 1.1: The cost of a bug at various stages of development, source Dark Views [101]

In order to uncover bugs in the early stages of development, tools have to be tested. Usually the better the coverage of the environment, the more bugs are triggered and can be fixed. To uncover the bugs, quality assurance teams and teams that focus on development of internal tools put a lot of effort into the design of new testing approaches. Nowadays, the majority of testing is performed automatically by advanced continuous integration systems (CI systems). However, there are still testing scenarios that cannot be automatically tested. The human element cannot be omitted in the process of testing.

Closely related to the problems of testing are the problems of releases and release cycles.

Because automatic testing can be triggered only once a build is finished. Nevertheless, there can be unit tests which, for example, can be triggered during the build itself.

It is quite expectable that the pressure for a short time to launch a product on the market is increasing. This also means that the time for the build and automatic tests must be kept short in order to give a developer more time for design and implementation and to keep the time needed for testing as short as possible. This trend has been confirmed by a study carried out at Sauce Labs [56]. The results are presented in the form of a white paper, which brings to light several interesting facts. Fig. 1.2 shows the frequency of build deployment.

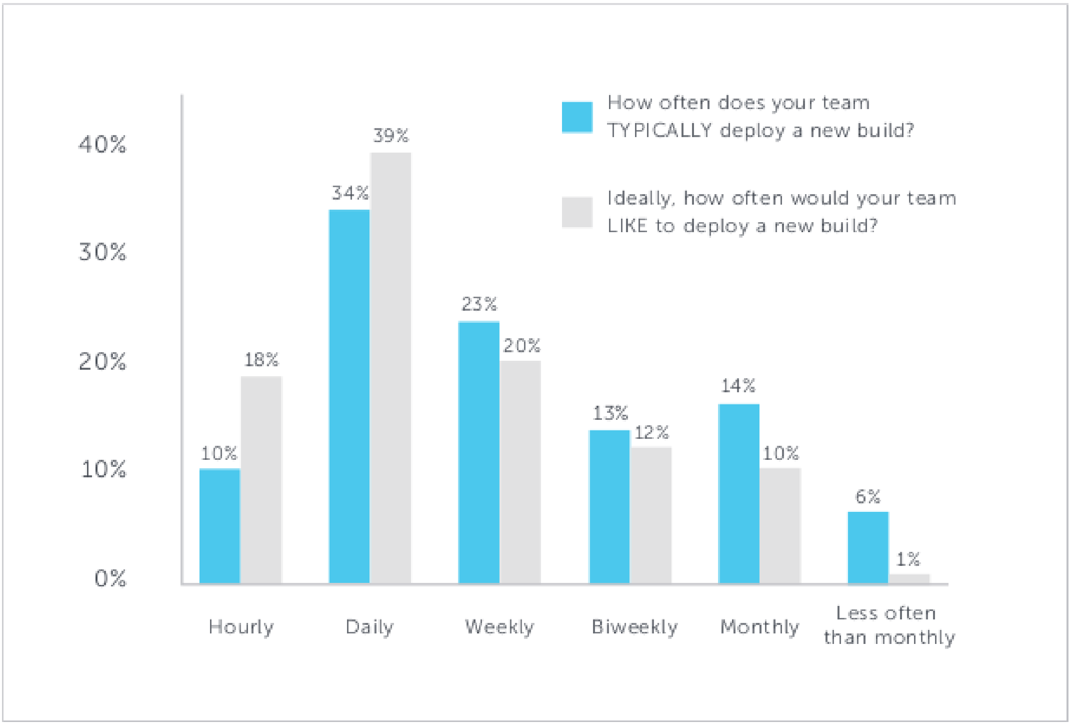

Figure 1.2: Frequency of build deployment, source Sauce Labs [56]

Only 10% of users are able to deploy the build hourly, but nearly 20% of them would like to deploy as often as hourly. The majority of users, nearly 35%, are able to deploy the build daily. Nevertheless, the percentage of people who would like to deploy it is nearly 40%. This proves that the need for a fast building system in the development cycle is crucial. CI systems play a very important part in the build automation and speeding up of the whole delivery process.

Closely related to the speed of the delivery process is the testing automation. Because, in cases when a developer has to wait for a long time for a build creation, it is usually faster to do the testing by hand, especially in cases where the testing scenario is not long or difficult. Some unit tests usually fall into this category.

A testing system, especially one for a complicated integrated development environment, such as a tool for hardware software codesign [24], must be capable of testing the separated parts, but also must be able to perform integration tests. In the last few years, an enormous amount of effort was invested into the testing environments. All the main development languages have advanced testing frameworks. To mention some of the biggest ones, I should name Selenium [88], Arquilian[11], Cucumber[21] and Autotest[13].

However, according to Sauce Labs, the majority of testing still has to be performed manually, or with a small amount of automation as is demonstrated in the following Fig. 1.3.

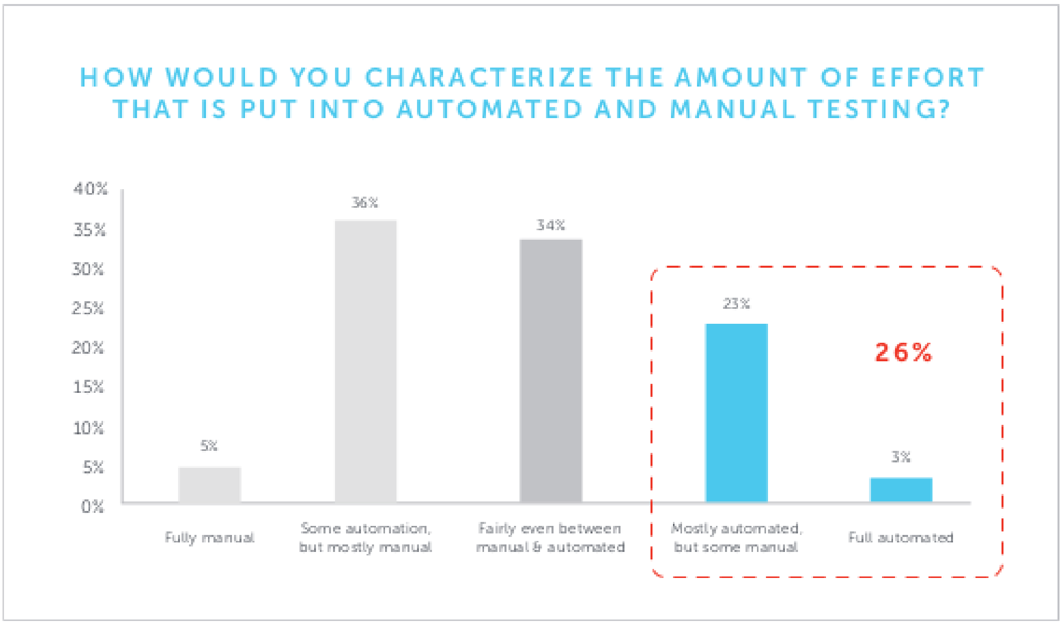

Figure 1.3: The portion of automated testing, source Sauce Labs [56]

There is a pending question of what lies behind such a small percentage of fully automated and mostly automated testing. Because only 26% of automated testing in total is definitely not an encouraging number. One of the reasons can also be the time pressure.

1.2 Problem statement

The current extremely competitive market of electronics of all kinds is very sensitive to the time it takes to introduce new products. Errors in design and implementation of a product, not only increase the cost of the final solution, but also cause delays that, in the end, mean a financial loss.

This drives the demand for fast and efficient testing systems. These testing systems must tackle several challenges:

- to provide a high level of automation of the testing procedure,
- to restrict the time needed to discover an error, this includes a fast rebuild of all tools that are needed for testing,
- to clearly identify an error and provide adequate information about the error,
- to define clear metrics to measure the progress of the testing process.

There are various types of errors. The types of errors that are usually discovered in the tools are logic and functional errors. Very often the developer misses a declaration or wrongly spells a name. Fortunately, this kind of error can be easily discovered. Also it is very simple to define the metrics for a successful build. The first two points are more challenging.

1.3 Tools for hardware software codesign

This thesis is going to discuss the area of testing hardware software codesign [25]. The hardware software codesign deals with the design of new embedded systems. Such a kind of systems can be found in a wide variety of devices, such as network routers or printers.

Embedded systems consist of one or more application specific processors (ASIPs). Each processor usually takes care of a single specific task and is, therefore, highly optimized for this task. The optimization is also the main difference from general purpose processors, such as the x86 family, which have to take care of various tasks.

The production of ASIPs in 2015 formed over 98% of the overall processor production. Therefore, this area is extremely important. Technology used for the creation of any ASIP is called System on the Chip (SoC) [87]. Such a technology allows integration of several ASIPs on one chip together with peripherals, such as memories, busses and others.

The development of current ASIPs must be done in a very short period of time [99]. In order to do so, it is common to use tools for the hardware software codesign. A hardware description language (HDL) is allways in the core of such tools. The development is done in a modern integrated development environment (IDE) that allows the designer to generate all the necessary tools, such as a compiler, an assembler or a simulator [83]. Then it is common that the application can be compiled in the same environment and simulated. These tools enable the Electronic Design Automation (EDA) and sometimes are also called the EDA tools [102]. Into the category of EDA tools falls, for example, the Processor Designer [97].

This kind of development environment shortens the development time significantly. However, each piece of software contains errors, and environments for the hardware software codesign are not an exception. Some of the tools are more error prone than others. From my point of view, the most critical is the compiler. Because if there is an error in the compiler, the compiled program might not work properly. Nevertheless, there are other parts, mainly the SDK tools, which are also critical.

1.4 Testing of tools for hardware software codesign

Each part of software needs to be tested. In the case of such a complex tool as the hardware software codesing environment, the testing techniques should be very advanced and ensure thorough tests of separate components as well as integration tests. In this thesis I will focus mainly on testing of the toolchain and particularly on tests of the compiler, as the compiler plays a key role in programming of an ASIP.

Because the compiler is partly generated, I will also look at the process of generation. Errors that may occur during the generation process may in certain cases also affect the quality of the compiler and its backend.

The compiler is also used from various environments. Therefore, integration tests are also needed to ensure that the compiler will work correctly and independently of the environment from which it was triggered. Also the compiler plays an important role in other areas, such as verification or during the tuning of the design and so on. Overall it can be said that the role of the compiler is unique during the development of an ASIP. If we take into account these wider consequences, it can lead to a higher stability of the compiler.

1.5 The structure of the thesis

The thesis is divided into nine chapters. The chapters are organized in the following way.

The second chapter is called State of art and gives an overview of architecture description languages, as they are used for the description of the core, and from this description the tools are generated. It also describes retargetable compilers, together with the testsuites and generators that are used for compiler testing. The continuous integration systems are also part of this chapter.

The third chapter describes the Lissom project. It is targeted at the description of the toolchain, the software development kit (SDK), the way it is generated from the description in the ADL.

Chapter four bears the name Goals of the thesis and there are outlined the results I would like to achieve in the thesis. The following four chapters are dedicated to the solution.

The fifth chapter is devoted to the porting of the library. It describes the role of the library in the toolchain, the process of porting and also automation of the porting process.

The sixth chapter discusses the problems connected to the scheme of test selection. As I use tests from a large number of sources, I need to deploy an efficient test selection mechanism. In the chapter I describe such a method and also the way how to automatically generate files that take care of test selection.

Chapter seven focuses on the area of testing via a continuous integration server and also acceleration of such testing. This chapter introduces an improvement in the flow of testing jobs that brings significant time and space savings.

Chapter number eight is the last of the sections that are focused on the solution of testing problems. It deals with problems connected to the generation of testing jobs, describes the design and implementation of the generator of the jobs.

Chapter nine concludes the thesis. It gives the summary of the results, describes the utilization in the industry, the advantages and disadvantages of the chosen solution. At the very end of the thesis, the future work is also discussed.

Chapter 2

State of art

Nowadays, demands for applications are very high. For example, network routers have to process a very high number of packets every second. General purpose processors (GPP) cannot handle such demands. Moreover, there are often restrictions regarding the area that can be occupied and power consumption is also usually limited. GPPs cannot meet such restrictions.

2.1 Application specific instruction set processors

However, there is a group of processors that can meet such demands. Application specific instruction set processors are designed especially for these purposes. These processors are optimized for one specific task, so they can perform it faster, consume less energy and cover a smaller area. The cost of this is the inability to perform general tasks. Such systems are used, for example, in hand held devices. Such devices are issued every year. Therefore, there is a strong need for the ASIPs to be designed easily so that the time to market it is very short. However, this pressure results in shortening of the design time, as well as the testing phase. One of the goals of this thesis is to design a new approach to the testing of the tools that are used for the design of new ASIPs as well as testing of the tools that are used for programming of ASIPs.

2.2 Application specific instruction set processors design tools

There are two basic approaches to the deployment of a new core in embedded systems. Either it is possible to try to customize an existing design, so the designer uses the existing core and adds some accelerators to reach the desired performance or they try to develop their own solution. The second attitude has been more and more common in recent years, as the usage of the existing core is often subject to a charge. Modern complex integrated development environments are usually used for the development of a core from scratch. An architecture description language is very often in the core of such IDE. Some of the environments offer design tools together with verification, such as the product from Cadence. The ASIP designer from Synopsys is more focused on the architecture and toolchain development, but offers a hardware generator and verification as well.

2.3 Description languages

Description languages are closely related to the ASIP design. As some of the tools that are going to be tested will be generated from the description in the languages, it is necessary to have at least basic knowledge of them. There are two basic approaches to the description of the ASIP. One has a lower level of abstraction and the other one higher.

2.3.1 Hardware description languages

The first category are the hardware description languages (HDL). Languages such as VHDL[100], [12] or Verilog [33] can be put into this group. These languages have the capability to describe the processor, but on a very low level (the abstraction is very low). This set of languages requires the designer to have deep knowledge of the given area and to go to the very details of the design. Changes in the later stages of the design are very time consuming and the development also takes a lot of time. Moreover, these languages are suitable only for description of the hardware. These languages do not contain information for generation of an assembler or a compiler. Partly because of this drawback, another family of languages appeared and it is becoming very popular.

2.3.2 Architecture description languages

This group is called the Architecture Description Languages (ADLs) [78]. The ADLs use a higher level of abstraction and so they allow faster modification of the microarchitecture at all stages of the development. From a description in the ADL, the majority of tools can be generated. The ADLs do not usually contain such details as the HDL. There are two basic approaches to this problem. Either the details are computed automatically or the ADLs have constructions for a higher level of abstraction[82]. The Architecture Description Languages can be divided into three categories.

1. *ADLs focused on processor architecture* - This type of description languages usually offers a lower level of abstraction. This is caused by the fact that we need to describe various characteristics of a wide range of processors. A processor is typically represented as a set of functional units. A hardware description can be very easily generated from a description in this kind of languages. The problem in this case is with the instruction set representation. The information about it is not explicitly contained in the description. So the tools needed for the programming of the processor, such as an assembler or a compiler, cannot be automatically generated. Into this category fall languages, such as MIMOLA or MESCAL [69].

2. *ADLs focused on instruction set* - This family of languages is mainly focused on the description of the instruction set. It usually has a higher level of abstraction as we describe the instructions. The processor microarchitecture is not described at all. This means that the hardware description cannot be generated from such languages. Therefore, a designer chooses such languages in the case when he or she targets the software development. To mention just some of them, we can name nML [75], ISDL or CSDL[37]. For example, nML has very nice formalism for the instruction set design.

3. *Mixed ADLs* - The category of mixed architecture description languages is a mix of the two previous categories. This group of languages allows a description of the instruction set as well as a description of the microarchitecture. The description of the

microarchitecture is usually optional. The design typically starts by the description of the instruction set and later on, once the design is stable enough, the microarchitecture is described. If a model contains both descriptions, it is possible to generate the programming toolchain as well as the hardware representation. The representatives of these languages are, for example, RADL [92], EXPRESSION[42], LISA [44], ISAC [70] or CodAL [19].

2.4 Retargetable compilers

In this section, I will have a closer look at compilers, as the testing of compilers is going to play a substantial role in my thesis. First, I should explain what a compiler is. The compiler is a tool that takes a program in one language, in my case in a higher programming language such as C, and transforms it into another language, such as the machine code. There is a special class of compilers that are called retargetable compilers. The retargetability means that the compiler is able to generate from the source language code the target code for more than one target architecture. The compiler possesses this ability when it is able to do so either without modifications or only with slight modifications.

According to the [60], a compiler can be classified as one of the following:

1. *parametrizable compiler* - in this case the machine description consists only of numerical parameters and subtarget settings,

2. *user-retargetable compiler* - in this case the external machine description is given in a dedicated language, which contains retargeting information, the specification does not require in-depth compiler knowledge,

3. *developer-retargetable compiler* - in this case the target architecture description is also mostly in external files, but its specification requires extensive compiler expertise.

Compilers of the first category, *parametrizable compilers*, usually enable the user to choose between subtargets, such as various descriptions of the instruction set of a single processor.

The second category, *user-retargetable compilers*, are those generated from the ADL. A generated compiler falls into this category. The compiler generator is able to parse the description in the ADL, usually after some preprocessing, and with minimal user interaction generate the desired compiler that is able to produce the code for given target architecture.

The last category are the *developer-retargetable compilers*. These compilers are the most common ones. The *Low Level Vitual Machine* (LLVM) and the *GNU Compiler Collection* (GCC) belong into this category. Although most compilers fall into this category, compilers are delivered by a third party, because the modifications that are required for changing the target are massive and, in most cases, the end user is not able to make them.

I am interested only in C/C++ compilers as it is still the most popular programming language for embedded systems. The GCC and the LLVM are definitely amongst widely used compilers. I will also have a look at other compilers that are available as open source projects and also at the commercial ones. All modern compilers have a common scheme of processing the code. The scheme is shown in Fig. 2.1.

As is apparent from figure 2.1, only the backend is platform dependent. The rest of the compiler deals either with the source code or with the internal representation of the program that are both independent from the target platform. This means that if the user

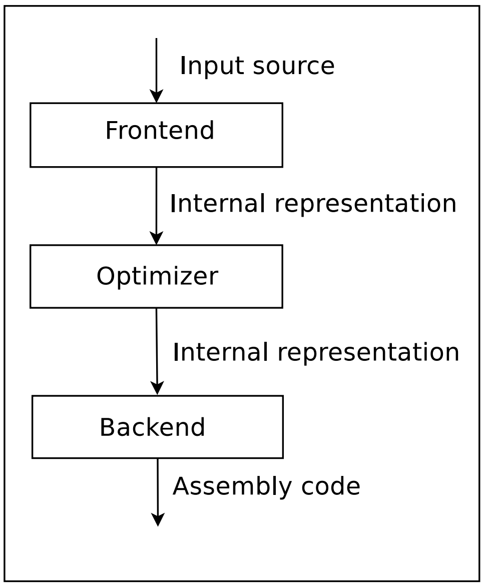

Figure 2.1: Scheme of a compiler

wants to create or modify the retargetable compiler, it is necessary to focus mainly on the compiler backend.

2.4.1 LLVM

First, let us have a look at the *Low Level Virtual Machine*, known as the LLVM [65]. The automatically generated compilers I am going to test will be based on this project, so I will describe it in a more detailed way. This, nowadays widely used, compiler started as a research project at the University of Illinois. The goal of the project was to deliver a modern compiler with a strong support for optimizations. It should also be capable of *Single Static Assignment* (SSA) based compilation [95] [47].

Since the beginning, the LLVM has gone a long way and nowadays it is a very large project that covers several subprojects. The most important one is the Clang, which has the role of the compiler driver and frontend of the LLVM. The LLVM project offers frontends for nearly all frequently used languages, such as Objective-C, D, Python, Ruby, C, C++ and so on.

The compilation scheme is shown in Fig. 2.2.

At the beginning, the input program is processed by the Clang frontend and in the intermediate form passed to the optimizer. After that comes the part which is unique for the LLVM. The LLVM compiler is able to link the modules in its intermediate form, and that is what makes it unique. This step is very useful for the testing as it enables the tester to give the author of the code the whole linked program in its intermediate form. After the linking, another optimization can be performed. The second optimization step is called the *whole program optimization* (WPO). After the WPO stage, the resulting program is either executed by the virtual machine or is passed to the assembler and later to the linker. This process produces the binary code, which can be executed on the target processor. Now I will have a closer look at the most important phases of compilation.

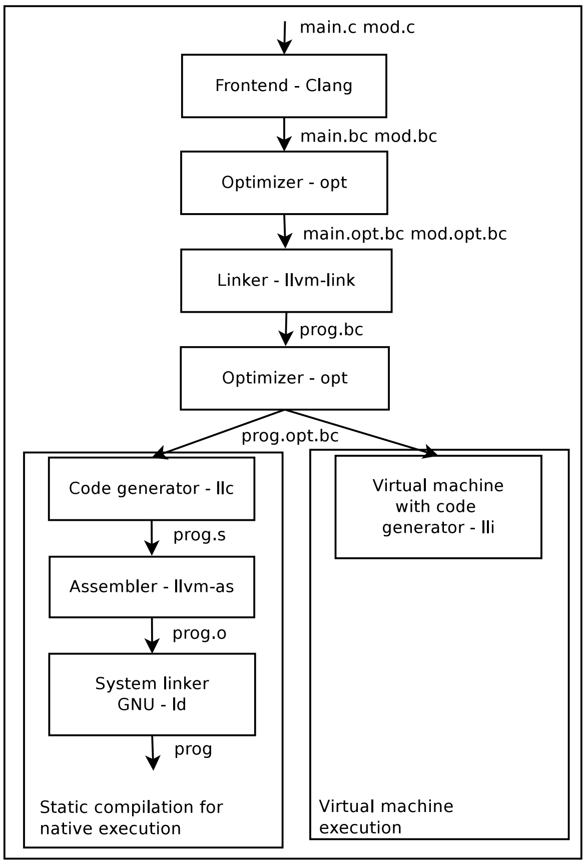

Figure 2.2: LLVM scheme

Clang

At the beginning, the input program is processed by the Clang. After this phase, the output program is in the form of the *abstract syntax tree* (AST) and has the suffix .bc. Several other phases work over this representation. Semantics checks and also static analysis are such phases. The static analysis can detect the first bugs in the source program. The semantic checks transform the input program into the *LLVM internal representation* (LLVM IR).

According to [55], the Clang is able to generate the code that is very similar to the one that was parsed. This ability makes the Clang usable for the source to source transformations.

It should be mentioned that the Clang project is quite a new part of the LLVM. As was mentioned above, one of the roles that Clang plays is the role of the compiler driver. Before the Clang took this position, another LLVM project was used, the project was called llvmc [66]. The llvmc offered a very basic configuration of the compiler driver, based on the suffixes of the source files. Each suffix was a node in a graph and by the edges the connections between the two nodes were created. There the llvmc has a special syntax for defining the behaviour of nodes and for interconnections.

The Clang project replaced the llvmc in the version 3.0. It offers far more sophisticated ways of defining how the compiler driver should behave. The solution is based on the C language. The Clang needs to know the so called target architecture triple. The Clang contains the table of the supported architectures that keeps the information about the data type sizes, endianess and alignment.

The Clang and the whole LLVM is a younger project than the *GNU Compiler Collection* (GCC). Before the Clang was added to the project, the role of the frontend was played by the *llvm-gcc* that generated the internal representation of the llvm. Majority of the projects that are compiled by the LLVM today were originally designed for the compilation by the GCC. This leads to the fact that the frontend tries hard to have a fully compatible command line interface with all the parameters. Nevertheless, the dominance of the GCC in this field is apparent and the LLVM project is always one step behind.

Optimizer

After the Clang phase, the code is optimized for the first time by the optimizer. The optimizer has the form of a pass manager. The manager schedules the sequence of passes based on the dependencies of each pass. The manager contains the dependency scheduler that does not always work perfectly. On the other hand, addition and removing of the passes is very easy and can also be tied to a command line option.

The optimizer works over the internal representation. The optimizer works with a target data layout [64]. The example of the data layout is shown below:

```
E-p:32:32:32-S64-n32-i32:32:32-f32:32:32-i64:32:32-f64:32:32
```

From this description, it is easy to get the endianess, the size of the pointer and registers and so on. It is the numerical description of the architecture. The first E specifies endianness, in this case the endianess, is big. The p:32:32:32 denotes that the pointer is 32 bits wide. The S64 means that the alignment on a stack is 64 bits. The following n32 says that the native integers are 32 bits wide. The following triples specify ABI and the preferred alignment for specific data types, such as int32.

Backend

The backend is the part of the compiler that is most heavily modified in our project. In the case of our project, the input of the backend is the program in the form of the LLVM IR and the output is the assembler. It is also the part of the compiler that mostly generates the errors. Because the output of the backend is the assembler, it is the most target dependent part of the compiler. The most important transformations, which are performed in the backend, are the following:

- *Lowering* - The key task is the transformation of the LLVM IR into the *direct acyclic graph* (DAG) [8].
- *Legalization* - The main role of the legalization phase is to replace the operations, which are unsupported at the target architecture by the equivalent sequence of the supported operations and, in certain cases, use a call of the runtime library, such as *compiler-rt.*
- *Instruction selection* - This phase works over the DAG, that keeps the LLVM operations and transforms it into another DAG, which contains the target architecture instructions. It can be said that it must map the instructions of the target architecture to the LLVM operations.

- *Register allocation* - So far the LLVM has worked with virtual registers. In this phase, virtual registers must be replaced by physical ones.
- *Prologue/epilogue insertion and frame finalization* - In this phase, the prologue and epilogue are inserted and the frame is finalized.
- *Scheduling* - This pass is quite simple and serves as the linearization of the DAG.
- *Assembly printing* - The final phase is the assembly printing. The IR is printed into the assembly file with all the necessary data definitions and other information.

The the most important phases that are computed in the backend are mentioned above.

Internal representation of the LLVM

I have mentioned several times the internal representation of the llvm called LLVM IR. Now we shall have a closer look at it.

First, I should mention that LLVM IR is based on the *Single Static Assignement* (SSA) [22]. The SSA denotes that each variable must be assigned only once. Various analyses and transformations use the SSA form. Thanks to the SSA the description is very straighforward and easy to read.

The LLVM IR is used through the whole compilation process. Thanks to the SSA form, it is type safe and is able to represent clearly the high level languages. It is also flexible and provides all the necessary operations. It is quite unique that LLVM IR supports integers of arbitrary bit width. No other internal representation allows this.

It also supports an unlimited number of virtual registers. This is given by the fact that it uses the SSA form. However, this can be a double-edged knife. I have experienced several times, during the testing of the generated compiler, that the architecture ran out of physical registers as the register allocator was unable to map the unlimited number of virtual registers to the physical ones.

The LLVM IR contains the following operations:

- *logical operations over integer and float*,
- *arithmetical operation over integer and float*,
- *conversions between the data types*,
- *comparisons*,
- *memory access operations*,
- *address computations*,
- *memory synchronisation*,
- *control flow handling.*

There are also possibilities for debugging in form of directives. What quite surprised me, is the support for the exception handling and special operations for garbage collection. On the other hand, it is not that surprising when I take into account that the architecture is virtual.

LLVM conclusion

I tried to sketch the infrastructure of the LLVM as the generated compilers are going to be based on the LLVM. The project is very well documented and together with the test-suite and related projects developed into one of the leaders of the open source compilers.

The fact that it is written in C++, makes the modifications easy. The majority of the codes are well structured and commented. The IR of the LLVM also contains a large amount of documentation. Thanks to this, a lot of companies are founding their solutions on the LLVM.

The project has usually two minor releases a year and this makes the migration of the user changes easily manageable, though not easy to perform.

2.4.2 GCC

The most widely used retargetable compiler these days is the *GNU Compiler Collection* (GCC)[103]. The project started in 1985 and the first release came in 1987. Nevertheless, the first stable version 1.x was produced four years later, in 1991. Currently, the latest version is 5.2. Nowadays, the GCC supports more than 40 architectures and is the leader in the field of retargetable compilers.

When I look at the GCC compiler I will find out that it roughly corresponds to the scheme 2.2. I can have a look at some of its parts. As it is not in the centre of my interest, I will not go into such details as in the case of the LLVM.

GCC frontend

The frontend of the GCC is in fact a collection of several different frontends. There are frontends for the majority of mainstream languages. To mention just some of them C, C++, Objective C, Java, Fortran and many others. The output of all frontends is the same intermediate language, which is called GENERIC. The reason for the intermediate language is simply to have language independent representation.

The representation looks very similar to the C language. In the following example, there is the C code and its representation in GENERIC without some details:

```
int res;
void sub(int a,int b){
res=a-b;
}
{
res=a-b;
}
```

GENERIC creates an interface between the frontend and the optimizer.

GCC optimizer

GENERIC description, which is used as an interface, is taken by the optimizer and transformed to the new internal representation called GIMPLE. GIMPLE is in fact the three-address representation of GENERIC. GIMPLE consists of tuples of exactly three operands. There are, of course, exceptions, such as function calls. It is possible to see here a correspondence to the LLVM IR, because GIMPLE is also SSA based.

The optimizer works over GIMPLE and performs target and language independent optimizations. GIMPLE is similar to GENERIC but it is more restrictive and simpler. It does not contain control flow structures, it is SSA based and three-address. For the example above, the GIMPLE representation is as follows:

```
sub(int a,int b)
gimple_bind<
int x.0;
gimple_assign<minus_expr,x.0,a,b,NULL>
gimple_assign <var_decl,x,x.0,NULL,NULL>
```

Once the optimization phase is finished, the code is passed to the last part of the GCC, to the backend. As the GENERIC IR was the interface between the frontend and the optimizer, GIMPLE is used as the interface between the optimizer and the backend.

GCC backend

The backend is the last main part of the GCC compiler. It works over the *Register Transfer Language* (RTL), which is the internal representation for this GCC part. The RTL is used through all the passes that are incorporated in the GCC backend. The form has been inspired by the Lisp language. The most typical features are nested parentheses that are used to indicate the pointers in the internal form in this case. There are two main blocks in the GCC backend:

- *Expand pass* - This is the first pass, the main aim of the pass is to create the RTL description from the GENERIC description. I can say that this pass generates the instruction list in the form of the RTL.
- *Target dependent optimizations* - Once the instructions are generated, the optimizations, which can be performed only over the target machine code, are performed. Into this category fall the peephole optimizations and so on.

The output of the backend is typically the assembler optimized file, which can be further processed.

GCC conclusion

Over the years, the GCC has grown into a very mature compiler. It is always at the fringe of development. It is an implicit compiler for the *(Berkeley Software Distribution* BSD system and is widely used in Linux distributions. Its frontend and libraries are up to date with the latest standards of C++[48], OpenMP [17] and so on. It also has the largest user base and the development is very active.

On the other hand, due to the extension of the project, it is difficult to maintain and modify it. The forms of the intermediate code are hard to read and the GCC also has a more restrictive license.

2.4.3 Other compilers

Apart from the two main open source projects, there are also other retargetable compilers that should be mentioned.

- There is a Target [96] project Chess/Checkers. The goal of this project is to enable the generation of the C compiler from the description in the nMl. [74]. There are articles about the backend generation from the nMl, but they are not precise enough.
- Another project is called Machine SUIF [94], [14]. SUIF is a research platform aimed at high performance compilers. The main focus is on loop transformations, scalar transformations and software pipe-lining. The platform consists of a small kernel and a tool kit for various analysis and optimizations. The main purpose of the core is to gather all necessary information needed for the other optimizations, to provide support for the IR manipulation and the provide the interface for other compilation phases.
- Very similar to SUIF is the Trimaran project. It is an extensible compiler framework by HP, which focuses on code optimization techniques. It supports mainly VLIW processors [59].
- CoSy compiler is a retargetable compiler developed by the Associated Compiler Experts [5]. The CoSy supports a very broad range of backends, from 8-bit controllers to the 256-bit VLIW architectures. The compiler consists of the frontend, the optimizer, which is architecture independent, and the retargetable backend. The backend can also be generated from the LISA language.
- CompCert is a very interesting compiler project. The main difference from the other mentioned compilers is the fact that it has a certified core.

Now after I have gained the idea about the compilers, I can move to another part that is closely related to compiler testing, the standard libraries.

2.4.4 Standard library

The language, whose compiler is generated, is based on the grammar that defines the syntax of the language. But the compiler itself is difficult to use. What makes the compiler really useful is the standard library of the language, whose compiler is generated.

That is true for majority of programming language. Because I am interested in the C programming language, I will have a look at the library of the C. The library for the C language is specified in the standard [9]. It is the subset of the C library POSIX specification. It is also called ISO C library.

In comparison to standard libraries of other languages, such as Python, the standard library of C is small. It provides only the basic sets of mathematical functions, functions for the conversion of types, basic manipulation for strings and file and console-based I/O.

When I compare the library with other language libraries, such as C++, Java or even Python, I find that it really holds just the minimum of functionality. Other language libraries provide, for example, containers, GUI tool kits or networking tools. The exact opposite of the C standard library is the Python standard library. The Python standard library provides, for example, clients and even servers for the common network services or multimedia services.

However, there is one big advantage of the small standard C library. It is the fact that in order to provide a working version of the library for a new platform, the amount of effort I need to expend is relatively small.

The main parts of the standard C library are the following:

- *Data types* - The data types provide the declaration of how the data are stored and what operations are permitted over the data.
- *Character classification* - In this section there are declared functions that are used for the test of the character membership, for example `isdigit()`.
- *Strings* - A set of functions that implements operations over the character or byte strings, such as a concatenation or a copy.
- *Mathematics* - An implementation of the basic mathematical functions for integer, float and other data types.
- *File input/output* - An implementation of many functions for the standard input and output. The function forms the main part of the `stdio.h`.
- *Date/time* - Functions that provide conversions between the date and time formats, a time acquisition.
- *Localization* - An implementation of the basic localization routines.
- *Memory allocation* - Dynamic memory allocation, the heart of the library, functions like `malloc, realloc`.
- *Process control* - A very important part of the library, basic functions for starting and termination of the process.
- *Signals* - Closely related to the process control, definition of the program behaviour when it receives the signal.

Some parts of the library are more error prone than others. There are certain parts of the library that are well known for overflows, such as `gets()`, and some of them are deprecated. Other functions are considered *thread unsafe.* None of these are crucial for the developers as there are always ways how to overcome such problems.

Even though there are several different standard C library implementations, the above mentioned parts are common for all of them. I will now have a closer look at the *Newlib library* as it plays an important role in the thesis.

Newlib Library

The Newlib library [77] is a collection of several parts that are all distributed under free licenses. It is the C standard library implementation that is intended for use in embedded systems.

The library is currently maintained by the Red Hat corporation [43]. The Newlib project is currently used in the majority of commercial and also non commercial embedded systems. It is particularly popular for the ones without an operating system.

The library has a strong support for porting (an addition to the new platform) and because of its popularity, there is a lot of documentation about the porting, for example [39], [15].

It is very well prepared for the addition of a new platform. It is divided into two parts. The first one is the `newlib` directory. It contains the majority of the code for the two main libraries `libc` (the core of C library) and `libm` (the mathematical library). Some architecture specific code might be found here.

The other part is the `libgloss` directory, called also `Board Support Package` (bsp), contains the platform dependent code. Therefore, during the porting mainly, the `libgloss` directory has to be targeted.

I will devote more space to the description of the porting in the next sections.

2.5 Brief history of compilers and testing

In this section, I would like to describe several important phases of the computer languages development and compiler testing. It is important to know the history of the compiler testing as it can give us clues that will help us in the current situation. The phases of the compiler testing are very closely connected to the evolution of the compilers itself.

2.5.1 First languages

The first programming languages were developed around 1950 together with the first electrically powered computers. The computers had limited speed and memory capacity. These computers were programmed in the assembly language. This required an enormous intellectual effort and was extremely error prone.

Therefore, there was a need for higher level programming languages. One of the first languages with real compiler was Fortran. The name was derived from *Formula Translation*. It dominated the area of scientific and engineering application for over 40 years. Another language introduced in this era was the LISP language.

At the very beginning, testing was performed manually, because computer time was extremely expensive. Nevertheless, even in those days, I can expect that there were first primitive test-suites. A great change came with the language *ALGOrithmic Language* ALGOL. This language introduced several innovations, such as blocks with `begin, end` and nested functions, which are still used in majority of modern languages. It was also one of the first languages that attracted attention to the formal definition of the language.

2.5.2 High level languages

At the end of the 1960s and the beginning of the 1970s, languages that are still in use today appeared. Most notably it was the C language. The language was developed in the Bell Labs as an imperative general-purpose programming language. For this type of languages, it is typical that they have test-suites. The most famous C compiler, the GCC project, started in the late 1980s and from the beginning it has had a set of testing programs. From a certain point of view, languages, such as C, are simple to understand and use. They are not object-oriented and do not contain other components that would make the testing more difficult.

The C language in the first version was very simple. The feature can be demonstrated also on the ACE test-suite. The tests for the later standards of the language form over 80% of the test-suite.

In that period, the test-suites were the main testing tool. The Smalltalk language was introduced in the same year as the C language.

2.5.3 Object-oriented programming

Smalltalk together with Simula were the leaders of the object-oriented programming. The C++ (also called C with classes) appeared later. Regardless of the object-oriented language, once the programmer could create the classes and their instances, i.e. the objects, there appeared a need for new kinds of testing.

Unit testing methods are focused on testing the classes internals. In the object-oriented programming, unit testing means testing of a certain class or the interface to identify whether they are fit for use. In unit testing, each test case should be independent from others. If the class needs to interact with other classes, mock objects or method stubs can be used. The unit testing should ensure that the objects behave as expected by the programmer. This area is very large. More information can be found in [104], [46], [68].

For example, in the article by Tao Xie [104], the authors propose a framework for differential unit testing. The main aim is to reduce the amount of manual work. The framework called *Diffut* uses the *Java Modeling Language* (JML). The Diffut generates the wrapper classes and automatically inserts annotations into the classes under test. The annotations invoke the corresponding method in the other version of the class and compare the return values.

The integration testing is aimed at testing of an interaction of the classes. It can also be understood as testing of various modules that are combined together and the modules are tested as a group. This kind of tests is usually performed after the unit testing is finished. The purpose of this test is to ensure that the modules can co-operate in the expected way and that they have the desired reliability and performance. More information can be found in [16], [35].

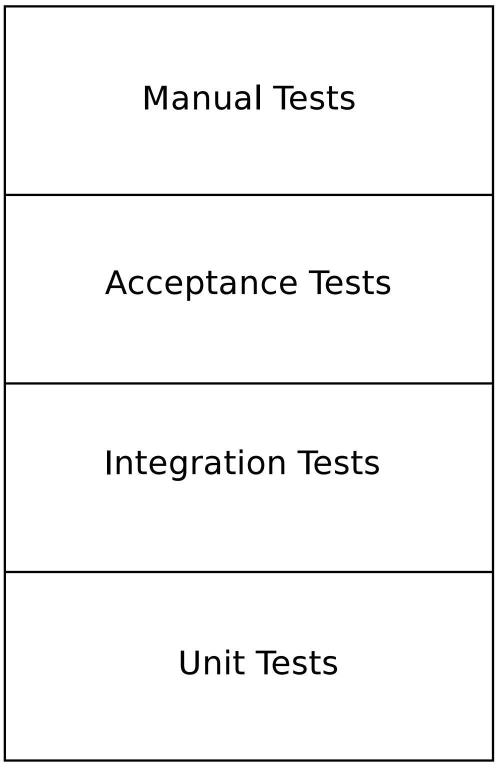

Figure 2.3: Scheme of testing

So far I have described only two types of testing that are usually mentioned. The common testing scheme deploys the following types of testing:

- *unit testing*,
- *integration testing*,
- *acceptance testing*,
- *manual testing*.

The acceptance testing [23] is usually performed by the customer. This type of testing is conducted by the customer to decide whether the requirements of the specification or contract are met. It should provide an answer to the question if the speed, area, power consumption of the provided chip are in line with the specification, when I talk, for example, about the ASIPs.

Manual testing is the last of the testing methods. Even though we posses the most advanced technology, some of the bugs can be uncovered only by human intervention.

Viewed from the other perspective, I can say that all of these types of testing are not specific for software development. It is possible to find variations of this attitude also in other industrial branches, from mechanical engineering to the pharmeceutical industry.

For a very long time, development of new programs was performed without the *Integrated Development Environment* (IDE). The first IDEs came to life in the 1970s. During the 1980s the leader in this field was *Maestro*. The most widespread IDEs of today are *Eclipse* [36] and *Visual Studio* [73]. The first release of Visual Studio took place in 1997 and the first released Eclipse appeared in 2001. Eclipse is a plugin based solution. Nowadays Eclipse provides a plugin for all major languages.

The current IDE usually provide support for the unit and also integration testing. Both types of testing are triggered via different components, but it can be done from the environment.

Today developers have good experience with so called *Continuous Integration* (CI) servers. The main idea of the continuous integration [38] is to avoid integration problems in the later stages of development. Developers are encouraged to merge with the main development line several times a day and execute tests over the result and keep an eye on the integration continually.

The technique was first suggested by Grady Booch and was called the *Booch method*. Later, it was adopted in the extreme programming and resulted in integrating once or more times a day.

By certain groups, the continuous integration is not accepted as an improvement over the frequent integration. However, I would say, that the majority of the great software houses use the continuous integration servers in the development cycle. The CI servers can be easily integrated with version control systems and allow automated builds via various hooks. Therefore, the build automation is at a very high level.

Also the deployment of the builds is highly automated, when using the CI server. Because the servers allow running the scripts, the release can be automatically linked to the customer.

2.6 Compiler testing methods

Compilers play a crucial role in the development of any software project. However, the emphasis placed on the quality assurance is not high enough. Authors of the mainstream compilers very often prefer testing by users *in the field*. This approach is not necessarily bad. The advantages of such an approach are clear. The authors of the compilers can outsource the testing to the user. However, fixing of the issues must be done in-house and the resolution of the bug in the production phase is far more time consuming than doing so in the earlier stages of the development.

Although there are modern ways of a formal verification of the compiler, they are not very common. For example, the CompCert compiler disposes of the formally verified core [57]. The price for this approach is the fact that 3/4 of the code in the CompCert compiler are devoted to the verification techniques.

But this approach is not suitable for the mainstream compilers, such as the GCC or LLVM. The developers of such projects still rely on test-suites. These test-suites are run continuously. One of the main disadvantages of such test-suites is the fact that they are very often composed of test cases that were reported by users. This brings the minimal regression, as we always test for occurrence of already known bugs, on the other hand, there are no methods how to detect new bugs.

Uncovering of new bugs is targeted by another type of testing. The *random generators* are used at various levels for the detection of new bugs in compilers. Very common are generators of random programs for higher level languages, such as C or Haskell. But if we understand the compiler as a tool that takes a program written in a high level language, such as C, and creates the binary code, there is also a motivation for random testing in several sub-areas. One of these sub-areas is the assembly level. In our team, we use such a random assembler generator.

In the sections below, I will give an overview of the current situation in both of the above mentioned areas. First, I will focus on test-suites and later I will devote some space to the random generation tools.

2.6.1 Test-suites for the C compiler

As my work deals mainly with the C compiler, I will focus on the sets of tests that are designed for the C/C++ compiler. The majority of the big compiler projects, such as GCC and LLVM, are distributed together with compiler test-suites. But there are commercial test-suites, such as the *ACE test-suite* or the *Perennial test-suite.* Companies developing such testing sets are very well aware of the fact that compiler testing is a growing area. The standard techniques are not able to cover the needs of the modern compiler development.

The test-suites are mainly used for *regression testing.* The aim of regression testing is to ensure that the software does not contain bugs, which we have uncovered during the process of development. The GCC test-suite and also the LLVM regression tests are sets of tests written by the developers of the compilers. The bugs were either found by the authors or were reported by the users. By execution of this test set I ensure, that the already known errors do not reappear. But by this approach I am not able to discover new errors. Very seldom do the already written tests trigger a new unknown sequence that results in an error.

GCC test-suite

The GCC test-suite [40] is a part of the compiler from the early stages of the development. It is distributed under the same licence as the compiler and contains a vast number of tests, which is true for all the other test-suites as well. The GCC test-suite does not come with the infrastructure and has clear reports, once the testing is finished.

The test-suite contains various types of tests. There are tests for C as well as for C++. As we do not support the full C++ in our project, I use mainly the C tests for

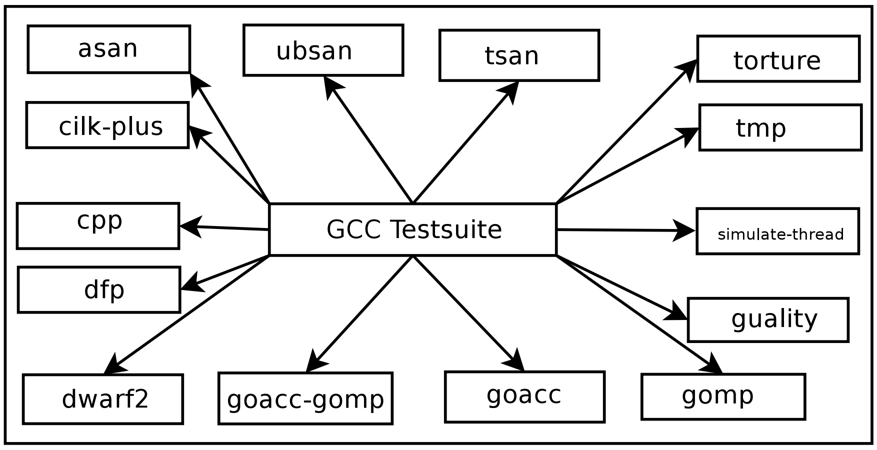

Figure 2.4: Scheme of the GCC directory structure

the testing. There are very simple programs, as well as larger programs, such as *SHA* or *Dhrystone algorithm*. The tests are very well sorted into directories 2.4. One of the greatest disadvantages of the GCC test-suite is the fact that the tests are not sorted. There is a certain directory structure, but it is very vague. For example, if a user wants to filter the float tests or tests that use only integers, they must do it by themselves.

The test-suite contains the torture part. These tests are meant to be compiled several times with different options. The torture test-suite is divided into several directories. Some tests are designed to be executed after the compilation but there are also tests that are designed only for compilation and should not be executed.

The disadvantages of this test-suite are very similar to the disadvantages of the compiler. The project of the GCC compiler is quite old and so is the test-suite. Moreover, the tests are usually only added to the test-suite. There are test cases that once triggered an error in the original code, but the code is no longer part of the compiler. Another problem is the fact that tests are not properly sorted and the test-suite does not contain an infrastructure. Although this can be viewed as an advantage, as I do not have to modify the existing code.

LLVM test-suite

From the LLVM project [65] there also comes a test-suite. This test-suite has two major parts. There is a regression test-suite and the benchmarks.

The regression test-suite is similar to the GCC one, which was described above. This part contains the test cases gathered during the development phases. The test cases are usually small pieces of code, which test a specific feature of the LLVM or trigger a specific bug. The language they are written in depends on what part of the LLVM is tested. The test-suite possesses a special driver for such tests, it is called *lit*. The directory, which contains the regression tests, is further broken into subdirectories that are named after the parts of the LLVM compiler that are tested by the cases contained in the given directory.

The other part of the LLVM test-suite, which in this case means benchmarks, is very different from the GCC test-suite. The LLVM test-suite is in fact composed of various benchmarks. The smaller programs meant for regression are kept separated. The rest of the test-suite, the benchmarks, are sorted into directories and thanks to the well designed makefile system the user can easily enable and disable the directories. In Fig. 2.5 is the directory structure of the LLVM test-suite.

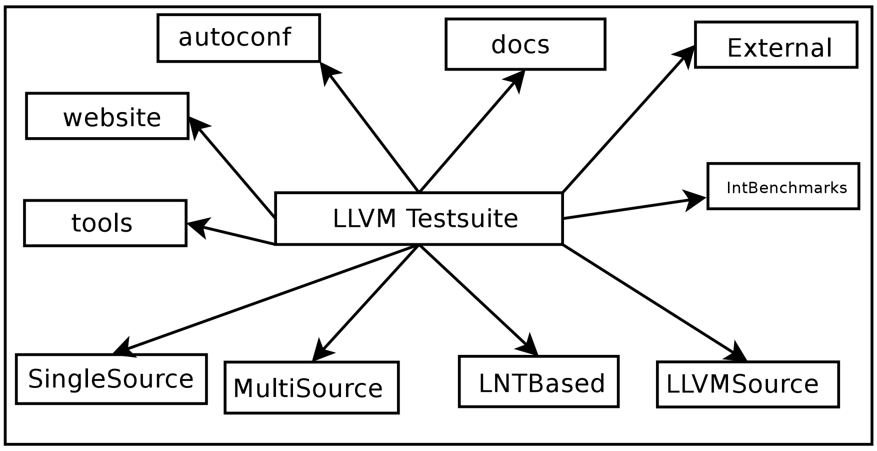

Figure 2.5: Scheme of the LLVM directory structure

The main directories, containing the tests, are named `SingleSource`, `MultiSource`, `IntBenchmarks` and `External`. The majority of the benchmarks lie in the directories `SingleSource` and `MultiSource`. The `SingleSource` benchmarks are usually smaller ones that are written in just one file and very often compute a certain value, and the `MutliSource` directory contains subdirectories with complex benchmarks and whole applications. The `IntBenchmarks` directory contains benchmarks that use only integer numbers and there is a special directory for benchmarks taken from external sources.

The `MultiSource` benchmarks very often use input and output into the files. This may cause several problems because results of such tests are typically evaluated by a comparison. The referential output and the output from our tool-chain are compared. In the case of text output, it may introduce some issues as the text output may have different format due to the debugging.

The format of results in the LLVM test-suite is very simple. The results show, where applicable, a comparison between the GCC and LLVM compiler. The basic metrics for the comparison is the runtime of the benchmark, and also the number of cycles that were needed for the execution, as can be seen in the following example.

```
Program                                             | GCC      LLC   |FLAG
IntBenchmarks/BenchmarkGame-fannkuch/fannkuch       |0.0320  38.6000|ok
IntBenchmarks/BenchmarkGame-recursive/recursive     |0.0200  74.2480|ok
IntBenchmarks/BitBench-five11/five11                |0.0160  16.5960|ok
IntBenchmarks/Dhrystone-dry/dry                     |0.0200  24.6960|ok
IntBenchmarks/FreeBench-analyzer/analyzer           |0.0280  70.6040|ok
IntBenchmarks/FreeBench-fourinarow/fourinarow       |0.0000  20.2040|ok
IntBenchmarks/McCat-01-qbsort/qbsort                |0.0160  22.6560|ok
IntBenchmarks/McCat-03-testtrie/testtrie            |0.0080  15.8400|ok
IntBenchmarks/McGill-exptree/exptree                |0.0000   0.0880|ok
```

```
IntBenchmarks/McGill-queens/queens             |0.0800 78.5160|ok
IntBenchmarks/MiBench-consumer-jpeg/consumer-jpg|0.0080 15.7000|ok
IntBenchmarks/MiBench-security-sha/security-sha |0.0040  5.6520|ok
IntBenchmarks/Shootout-ackermann/ackermann      |0.0040 13.9400|ok
IntBenchmarks/Shootout-fib2/fib2                |0.0080 17.8920|ok
IntBenchmarks/Shootout-lists/lists              |0.0240 50.4560|ok
IntBenchmarks/Shootout-matrix/matrix            |0.0160 58.4000|ok
IntBenchmarks/Shootout-methcall/methcall        |0.0120 21.6400|ok
IntBenchmarks/Shootout-nestedloop/nestedloop    |0.4920 20.4960|ok
```

The example shows results of several programs from the `IntBenchamrks` directory. In the first column there is the name of the benchmark followed by the runtimes of the GCC and LLVM. The program compiled by the GCC was executed natively, while the program compiled by the LLVM was executed on a simulator. The last column, called *FLAG*, holds information about the result. The example is not complete, as some of the columns were deleted, otherwise the results would not fit the page formatting.

Also in the case of a benchmark failure, the complete log files are kept in a specific directory. There are several logs which are kept from various stages of the compilation and also from the execution phase. The log files are very often quite large and it is not easy to identify where the problem is.

The system for the benchmark compilation is hierarchical. There is a system of makefiles which control the compilation as well as the execution of the benchmarks. Each benchmark can, therefore, be compiled and executed separately.

```
LEVEL = .

PARALLEL_DIRS = SingleSource IntBenchmarks

include $(LEVEL)/Makefile.programs

build-for-llvm-top:
	./configure --with-llvmsrc=$(LLVM_TOP)/llvm --with-llvmobj=$(
	   LLVM_TOP)/llvm --srcdir=$(LLVM_TOP)/test-suite --with-
	   llvmgccdir=$(LLVM_TOP)/install --with-externals=$(LLVM_TOP)/
	   externals
	$(MAKE)
```

In the example above, I demonstrate how easy it can be to enable or disable the directories. It can be done by a simple addition of the directory name to the variable `PARALLEL_DIRS`. Below the variable there is the command for configuration and build of the benchmarks. When I get to the lowest level of the directory structure, it is also possible to enable or disable any single benchmark. The system uses the standard configure scripts as well as a make program.

The system enables a parallel compilation and execution of the benchmarks, which keeps the speed of the testing at a very good level. The system is able to detect the number of cores and run the compilation and execution on several cores. However, due to the number and complexity of the test and also the fact that the tests run on a simulator, the testing is slower then I would expect.

When I look at the mechanism for the test selection, it gives the user a possibility to modify the compilation and execution of the benchmarks at will. But what is missing is the possibility to choose the benchmarks according to some predefined features.

Perennial test-suite

Apart from test-suites that come with the compilers themselves, there are a lot of commercial test-suites. The Perennial test-suite [81] is one of them. This test-suite contains its own system for the execution and compilation of tests. It also contains a special file that takes care of managing the input/output (I/O) subsystem. Thanks to this feature, it makes debugging of the failed tests a little bit more complicated as these files are linked to the test and one has to step through a large amount of instructions to get to the test itself.

The test-suite contains single source tests as well as tests that are composed of multiple files. The tests are sorted into the folders according to the C standard. The folders make the testing well-arranged as the user can exactly see in what part of the compiler there is an error triggered. The test-suite is easy to configure and all the files which have to be configured have a good documentation.

An example of the configuration taken from the configuration file is shown below. The explanation forms a part of the example, which shows the phase of linking. The user can specify all the formats and necessary tools and can also specify the files that should be removed after this phase. This approach gives the user a large amount of freedom for specification of every single phase. On the other hand, the configuration file is quite long and certain passages occur several times without modification.

```
# Compile-to-Executable (CX_)
# ===========================

# How to compile one C source file to an executable. The full file
  name is
# %f. The base name is %n. It must be linked with the scaffold
  object
# module which is %o.
CX_C             = %(CC) %i %f %o __LINKED_FILES__ -o %n.xexe

# Files to remove thereafter, executable file not included.
CX_C_RM          = %n.g
```

The test-suite is delivered in the form of a source code, which allows the user to modify it in case of need. Before the use, I had to compile the driver and create the configuration file. I took the configuration file as a template. It was necessary to generate certain parts, such as paths to the toolchain and the name of the compiler. The testing was performed simply by running the driver with the given configuration file.

```
./driver
```

Nevertheless, the test-suite is not designed for an execution on simulator. I need to check more things than the return value of the simulator, and the test-suite does not allow this without modification of the source code.

I used to use the Perennial test-suite and during the deployment I more often found failing tests in the GCC test-suite than in the Perennial test-suite. The problems, which were revealed thanks to the Perennial test-suite, were usually connected to the linker. On the other hand, the test-suite has a very nice system of notifications at its disposal and the user exactly knew which part of the compilation failed. The test-suite is also always up-to-date with the latest standards.

Once the test-suite is executed, it gives the results summary similar to the example below. Each run of the test-suite has a unique number and the results are stored in a directory, which bears a label with this number. The folder contains all the temporary files that were not deleted as set in the configuration file, the compilation and execution log for each test and file with name failures. The failures file contains names of all the failed tests together with the description of the failure. This is very useful for further parsing and sorting of the failed tests. I used this file to pack all the temporary files together with the source of the failed tests. The package with the temporary files helps the designer of the compiler significantly during the debugging phase.

```
    *******************************************
    *                                         *
    *           TEST CYCLE COMPLETE           *
    *                                         *
    *******************************************

120 total test files:
        110 test files pass
        8 test files fail
        2 test files unresolved
        0 test files uninitiated
        0 test files untested
        0 warnings were issued
```

The speed of the testing is very high. The speed in connection with the fact that all the temporary files are kept for further use makes the test-suite usable for debugging. On the other hand, the debugging is complicated by the fact that the tests are linked together with the internal code of the test-suite. What I personally consider to be the biggest weakness of this test-suite is the test selection mechanism. The tests are kept in simple lists. There is no possibility of dynamically changing the set of tests.

SuperTest compiler test and validation suite

The SuperTest compiler test and validation suite by ACE [6] is one of the market leaders in compiler validation. I will call this test-suite just the ACE test-suite to shorten the name. This test-suite offers tests sorted according to the C/C++ standards. There is a specific directory for the C89 and C99 tests and these directories are broken further down according to the paragraphs of the standard. So for example, the C89 standard has the following summary:

Total number of files: 7

Total number of tests: 7

Paragraph	Subject
3	Language
3.5	Declarations
3.5.3	Type qualifiers
3.5.4	Declarators
3.5.4.5	Array declarators

Each test case is contained in one file. The first character of the file name can be either `t` or `x`. The files starting with `t` are positive tests. These tests contain the correct C programs and, therefore, should be compiled successfully. The files starting with `x` are negative tests. For these tests, a diagnostic is expected and a compilation is expected to fail.

Moreover, this test-suite offers so called depth test-suites. These tests are focused on testing of the basic arithmetic. There are thousands of tests for various operations with data types. These tests are platform dependent as they depend on the size of the data types so the tests for 32 and 64 bits differ. There is a special naming convention for the depth test-suites. An example of such a name follows:

```
c24.l48.f32.d64.tar.gpg
```

The name encodes a data model [4]. It specifies the target with 24-bit characters, 48-bit longs, 32-bit IEEE floats and 64-bit IEEE doubles. The following list defines all types that can be used in the suite name:

```
c    char
s    short
i    int
l    long
ll   long long
f    float
d    double
ld   long double
sfx  short fixed
lfx  long fixed
sac  signed accum
```

This list is very useful for me, as I will try to find a way to automatically match this pattern and select the corresponding depth test-suite automatically.

The problem with this attitude is that when I am developing a new core, I may choose such a combination of data types that no depth test-suite is suitable for it. Also, there is a problem of an automatic detection of the data type of a new compiler, and when I have a larger number of cores, the time spent on the selection might be enormous.

The test-suite uses a special driver for the execution of the tests. This driver is called `valid`. There are several modes of execution. A user can execute either a single test or all tests that are kept on special lists. The compiler driver takes the configuration file as a parameter. The configuration file is platform specific and keeps information about the position of the toolchain, parameters of the compilation and execution. The configuration file is not as detailed as in the case of the Perennial test-suite. There is also another tool called the `leash` which can be used for the execution of a certain test with limits. It allows the user to set a limit for a time as well as to limit the size of the output. Below there is an example of the execution of the `leash`, as well as the execution of the driver.

```
valid -e 'one 3/0/1/tall.c' default.cfg

valid default

Collecting test set...
Running nulltest...
Creating libst...

C Validation started

[==================================================]100% (2567/2567)
```

```
leash -t 1m -o 1k -e 512 a.out
```

The execution of the valid means that only one file `3/0/1/tall.c` will be compiled. The command `valid default` will execute the basic set of tests. The execution of the `leash` restricts the execution of the file `a.out` to one minute and the output is limited to 1 Kbyte on `stdout` and 512 bytes on the error output `stderr`.

The program `valid` automatically detects the number of cores and performs a parallel compilation and execution. However, it compiles the source files in the directories named after the cores, so it does not keep temporary files from the compilation and execution.

As far as the result reporting is concerned, there is a directory containing all the log files and the list of failed tests. There are separate log files from the compilation phase and from the execution phase of testing. There is a separate log for each clause of the standard. Below there is shown the compilation log of one of the tests.

```
TESTING: suite/2/1/1/2/t01.c
    New-line character with an immediately preceding backslash

    Compilation succeeded

RESULT: 2/1/1/2/t01.c       PASSED
```

Scripts for the log separation are written in Perl and are part of the test-suite, so they are not difficult to modify. I deployed this test-suite for nearly a year in our project and the results were very positive. One of the drawbacks is the large number of tests, so the execution takes a long time, if not tested on a larger number of cores such as eight or more. The code coverage increased when the test-suite was used slightly.

One of the drawbacks of this test-suite is the fact that the test selection mechanism is based on a simple list. Though it is not a list of tests but a list of directories that should be included in the testing. The driver detects whether the directory exists and if yes, it executes tests in the named directory and all subdirectories. This gives the user freedom to modify what tests will be placed in the testing directories.

2.6.2 Test selection mechanism

One of the most important criteria for use of the test-suite is the way of test case selection. All of the test-suites that were mentioned had serious drawbacks as far as the test selection is concerned. There are certain test-suites that do not possess any testing infrastructure and test selection mechanism at all. The rest of the test-suites gives just very basic options of the test selection.

The test selection is usually based on a simple list of files. In certain cases, the list of files contains only the test name, but in other cases, it contains the whole path to the test from the given directory, which is typically the root directory of the test-suite.

In the second case, when the list contains the full path to the test, I used this information when the test failed to pack it together with the temporary files that were created during the translation.

Nevertheless, these simple methods of test selection cannot be used for my purpose. There is a large theory concerning test selection methods testing [34], [58] or [7]. The methods can be divided into three basic categories:

- *Coverage techniques:* This approach takes into account the code coverage. The coverable program parts are looked for and choosen.

- *Minimization techniques:* This approach is similar to the coverage one but a minimum set of test cases is chosen.
- *Safe techniques:* This approach is not focused on coverage, instead all the test cases that produce different output are taken into account.

Nevertheless, I need to focus on different aspects of the test selection mechanism. I do not need to keep the set minimal. An important role here is played by the information about the instruction set that the compiler possesses. Very often the model from which the compiler is generated can dispose of a specific bit width. For example, I can create a compiler for the 16-bit model or for the 32-bit model. This characteristic influences the set of tests that can be compiled and executed. There are also other factors, such as the presence of the C compiler library and the presence of compiler-rt library and so on. All of these factors must be taken into account.

My test selection mechanism must be able to address such differences. I need to easily choose the test for each platform according to the bit width and the presence of certain libraries. And, in certain cases, also to specify directly that certain tests should not be executed on the given architecture.

2.6.3 Random generators

Another way of compiler testing are *random tests generators*. It is definitely not an easy task to create a random generator. The generated test programs must have the correct structure which is accepted by the compiler. The majority of today's compilers use multi-stage processing. There might be even dozens of stages before the final compiled code is produced.

For example, the LLVM is a framework that allows an easy insertion of compilation phases. It is common to add the optimization, or any other phases, into the LLVM processing chain.

During the compilation, the earlier stages must be finished without errors. So in order to test the later phases, the generator must produce a code, which passes the earlier ones. The requirements for passing vary. It may be just lexical correctness of the program or the correct syntax. In later phases, where the semantics analysis is solved, the program must be type correct in the case of statically-typed languages.

The generation of the valid sentences for the given language is usually based on a formal basis. Also the use of templates is very common, especially for the generation of more complex programs with specific semantics.

As I focus primarily on the C language, I have picked mainly the generators that produce test cases in the C language.

Csmith

One of such projects is the *Csmith* [105]. The Csmith is a random generator of C programs that aim at hardening of all known compilers. The Csmith attempts to avoid the undefined and unspecified behaviour in the generated programs while the expressivity of the generated programs is at a very high level.

To do so, the Csmith deploys relatively complex program generators. This program generator uses various techniques to produce safe programs. First of all, the generator uses structural constraint to avoid unsafe behaviour. Then, in cases where the constraints would be too restrictive, it performs a static analysis of the already generated code fragments. By doing so, the Csmith determines whether the given operation is safe or not. Also it often inserts runtime safety checks into the generated code.

The test evaluation is done by using differential testing with the use of different options of the same compiler, or by the use of different compilers, or a combination of both. The scheme of testing with the Csmith is shown in Fig. 2.6.

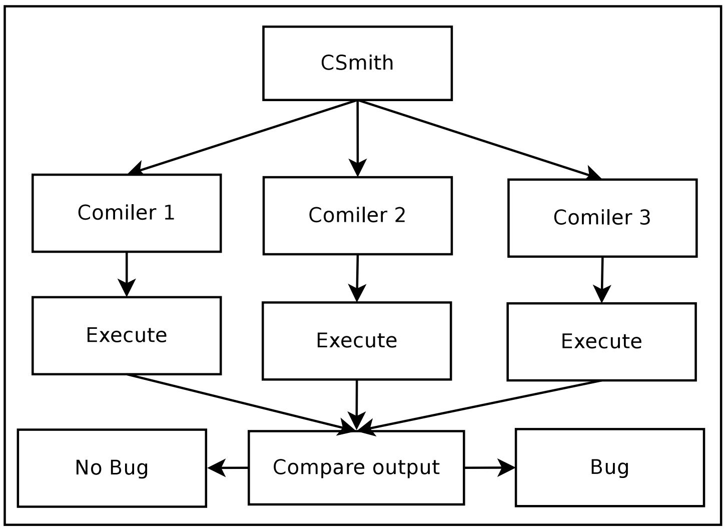

Figure 2.6: Csmith scheme

The evaluation of the test results is performed by the comparison of the checksums. The checksums are computed from the non-pointer global variables sampled at the end of each execution. The authors of the Csmith project used it for testing of various compilers. They tested frequently used projects, such as the GCC or LLVM, compilers with a certified core, such as the CompCert, and also commercial C compilers. The CSmith uncovered 325 bugs in these compilers, most of them in the GCC and LLVM. The CompCert that uses the formally certified core contained several bugs.

Test cases that expose a bug are usually not reduced as the reduction may introduce some undefined or unspecified behaviour that the authors try to reduce as much as possible. The programs that contain 8k-16k of tokens show the highest rate of bug triggering. The tests cases triggering errors are usually reduced by hand to get understandable test cases. One of such test cases is shown in the example below.

```
int foo (void){
signed char x = 1;
unsigned char y = 255;
return x > y;
}
```

This test case comes from the Csmith database [20]. It uncovered a bug in the GCC compiler that was shipped with the Ubuntu 8.04.1. According to the database, at all optimization levels the compiled program returned 1, while the correct result is 0. The compiler for Ubuntu contained several patches. The base version without the patches worked correctly. This situation is very common. Nearly all major distributions have compilers modified to suit their needs. This procedure very often brings new errors into an already stable product.

It seems that the greatest development of this project was around 2010. The Csmith is distributed under the BSD-style licence. The generated programs can be used for any other C compiler. The latest version of these tools is 2.2 and comes from the very end of the year 2014.

McKeeman project

McKeeman [71] has created a project that also uses a differential approach to the testing of the compiler but, unlike the Csmith, it uses inputs of various quality levels. This is something that was not possible with the Csmith. According to the article, the lowest quality level has a sequence of any ASCII characters, followed by a valid sequence of tokens and syntactically correct programs. The last level is presented by programs with well-defined semantics.

This method has been proved as a very efficient one in uncovering bugs in different stages of the tested compilers since this method tests the compiler in a complex way. What is very well-designed in this tool is the way it generates new test cases. The user can choose the level of generation. Whatever level is chosen, new test cases are created from the actual one by introducing small changes into the test case. If the new test case causes the compiler to crash, it is easy to track the bug.

Quite uncommon is the way which was chosen for the creation of the generator. The generator is represented by the Tcl script which is based on the context free-grammar based generator. This solution is enhanced by support of the context sensitive features, such as defined variables tracking. Each grammar rule has a weight. The termination of the algorithm is ensured by assigning small weights to the recursive rules.

The generated test cases are given to several compilers, as is usual for differential testing. If a test case causes an error, the process of shrinking is used for the reduction of its size. The test cases might have several hundreds or thousands of lines. As I have seen in the case of the Csmith, quite long tests have the highest rate of triggering of bugs. Very often the test cases can be shortened to several lines of code. Nevertheless, this can take up to tens of thousands of compilations.

The discrepancies between two of the compiled versions are handled in the following way. All potentially dangerous operations are replaced by error checking variants. I can demonstrate this feature on the example below. For example, consider the generated case that might be potentially dangerous because of the violations:

```
a << b
```

This example is replaced upon regeneration by the following:

```
int_shl_int_int(a, b)
```

The function checking function has the following syntax. It checks the integer shift out of range:

```
int int_shl_int_int(int val, int amt) {
assert(amt >= 0 && amt < sizeof(int)*8);
return val << amt;
}
```

The program is then re-executed. If the program execution fails, it means it contains an error, and the program is discarded.

I have already noted that the highest level of testing consists of programs that have well defined semantics. The programs are generated from specific templates at this level. These templates ensure that the generated programs have a certain high level structure and, therefore, keep certain semantic properties. However, I did not discover how difficult it can be to add such a template to widen the pool of the generated programs. The diversity of the generated programs is weighted against the semantics correctness. The generated test cases are very specific in comparison to the tests cases that have lower quality.

Quest

This project was created by Lindig [62]. The aim of this project is to create a simple tool for testing of the calling convention of the C functions in the C compilers. Random programs are generated containing C functions, which perform consistency checks to verify that the arguments were passed correctly. The types of functions are chosen randomly, and the body is then algorithmically generated. The generated cases are not usually very long, an example of a generated test case is below:

```
1   #include <stdarg.h>
2   #include <assert.h>
3   union A {float a; double b;}
4   c = { 52.54 };
5   struct B {double d; int e;}
6   h = { 78.01, 834 };
7   union C {short int f; char g;}
8   i = { 68 };
9   struct D {char j; double k;}
10  n = { 'c', 31.01 };
11  struct E {long long l; double m;}
12  o = { 167L, 17.2 };
13  union A
14  callee(struct D a, struct E b, ...)
15  {
16  va_list ap;
17  struct B x;
18  union C y;
19  va_start (ap, b);
20  x = va_arg (ap, struct B); /* 3rd */
21  y = va_arg (ap, union C); /* 4th */
22  assert (y.f == i.f);
23  /* fails */
24  va_end (ap);
25  return c;
26  }
27  int main( int argc, char **arg ) {
28  union A r;
29  r = callee (n, o, h, i);
30  return 0;
31  }
```

The GCC 3.3 compiler on MacOS X 10.3 passes `union C` incorrectly to the variadic function callee. The assertion in line 22 fails. This test case code was generated by the Quest tool. In any variadic function, extra arguments must be accessed using macro `va arg`, which receives the type of argument and returns its value.

The Quest generator found errors connected to the calling sequence in five different compilers. Moreover, the bugs were triggered by a very simple code. One of the explanations can be that the test-suites, which are used and also very often written by the compiler writers, contain only a very limited subset of combinations of the arguments.

Haskell generator

In the thesis by Palka [79] the generator for Haskell compiler is described. The generator described in Palka's thesis is able to generate only a subset of the Haskell language. But even generation of this subset was able to find interesting bugs in the compiler. Once the test case uncovering a bug was uncovered, the shrinking mechanism was used to reduce the size of the test case into the form that is suitable for the bug reports.

The example of the shrinking phase is in Fig. 2.7.

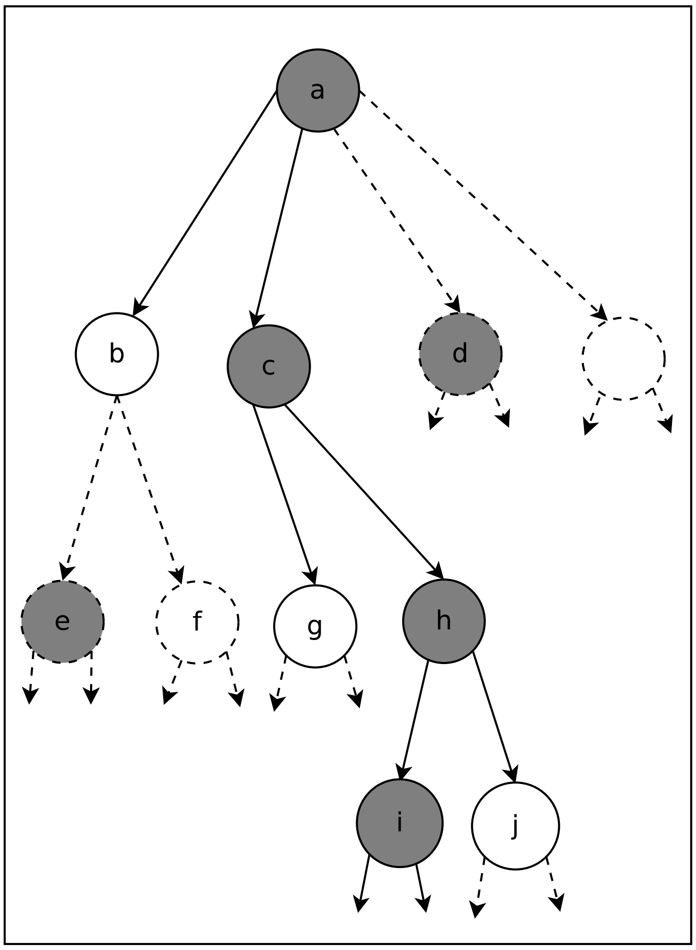

Figure 2.7: Shrinking scheme

The test cases that fail are marked as grey. The dotted test cases are not taken into account.

The shrinking mechanism is simple. Test case a is the original test case. This test case is further shrunk. The first shrinking step considers *a's* shrinking candidates *b, c, d* and so on. These test cases are put under test. Test case *b* is considered first. Unfortunately it succeeds and is discarded. Test case c fails, so it becomes the *current* shrunk test case. The shrinking candidates of c are tested in the next step. In this step, the candidate h is found to fail and the process continues with its shrinking candidates. The process terminates when all current shrinking candidates succeed, or when the test case does not have any, and the last failing test case is reported.

In this case the differential testing is used to uncover bugs. The approach of compiling the test case by one compiler with different optimization levels is used here. Also the alternative way of the differential testing was used where equivalent programs are used and the behaviour is compared. The second attitude was able to uncover more bugs.

However, in the case of the Haskell generator as well as in the majority of other examined random generators, the testing cannot be fully automatized. A lot of time is usually spent on reducing the test cases. This can be done automatically only for certain cases. When reporting a bug, the test case must be as clear as possible, and usually the automatically generated tests are not in the appropriate form.

According to the author, effective testing is dependent on spending effort on creatively devising properties. Also, a lot of bugs were uncovered by properties that were originally developed for a completely different purpose. This finding correlates with the statement that was made about test-suites. To uncover new bugs it is necessary to think about problems from another perspective and also some functionality in a different way.

But what is common for all the random generators is the fact that working with any of these tools gives the author, or even the user, a deeper understanding of how the compiler works internally.

Other generators

There has been one interesting bachelor thesis at the Faculty of Information Technology, Brno University of Technology. In this thesis [76], the author suggested a method of generation of sentences for the C language. The system consists of three parts. See Fig. 2.8.

The first and most important part is the generator called *Spitter*. Although the system is designed in a way that it can deploy any input, the generator is the heart of the whole system. The Spitter produces sentences of the C language.

The sentences are easily compilable and also the runtime is usually short. The compilation, as well as execution, are performed by a module named *Builder*. The Builder module not only keeps all the logs of the execution and compilation but it also keeps information about the way in which the program finished.

The last module is called the *Comparator*. This module is responsible for deciding whether an anomaly has been discovered or not. If so, all the logs and temporary files are kept for further debugging. The project mentioned in the thesis was successful in finding bugs in the known compilers.

Nevertheless, this project also suffers from serious problems. Although the generated programs do reveal problems in the compiler, the analysis of the test case is usually very difficult. What is even more disturbing is the fact that this dynamic method produces a lot of false positive test cases. This points to the fact that the Comparator module should be improved.

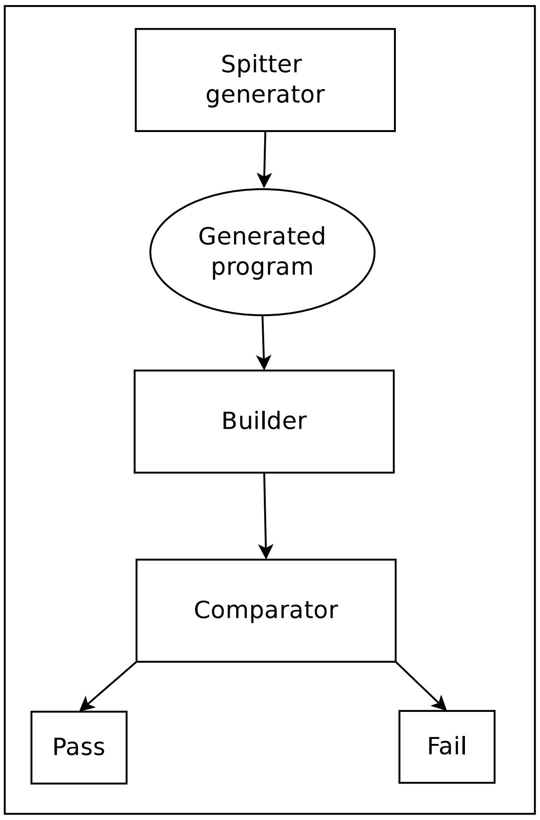

Figure 2.8: Spitter scheme

2.7 Continuous integration

As was mentioned above, the continuous integration servers are nowadays used for deployment and testing of new packages and releases. Before the continuous integration method was deployed, the development of software had had to deal with several serious disadvantages. The teams of developers merged the code together via non systematic methods and they were very often forced to rewrite certain parts of the code. A process like this very often took weeks and sometimes even months. This very often led to inevitable delays in the process of development [67].

Nowadays, we use modern tools for the process of software development, these make the whole process faster and easier. Because today the software development is not only the coding but also continuous testing, version control of the code, quality assurance and observation of metrics. Continuous integration tools make this process faster, less error prone and they also help with automation of certain parts. It gives the programmer a powerful tool for error detection and also reporting of errors, and it also helps with the release management.

However, the utilisation of the continuous integration processes Fig. 2.9 bring certain restrictions. The process of building must be automatic. This means that it should be reliable and without user interference. When a programmer saves the changes into the version control system they should be sure that the code is compilable. Also the fix of the broken code is a part of the development process and should be made as soon as possible. Testing is one of the fundamental parts of the development cycle, and the regular triggering

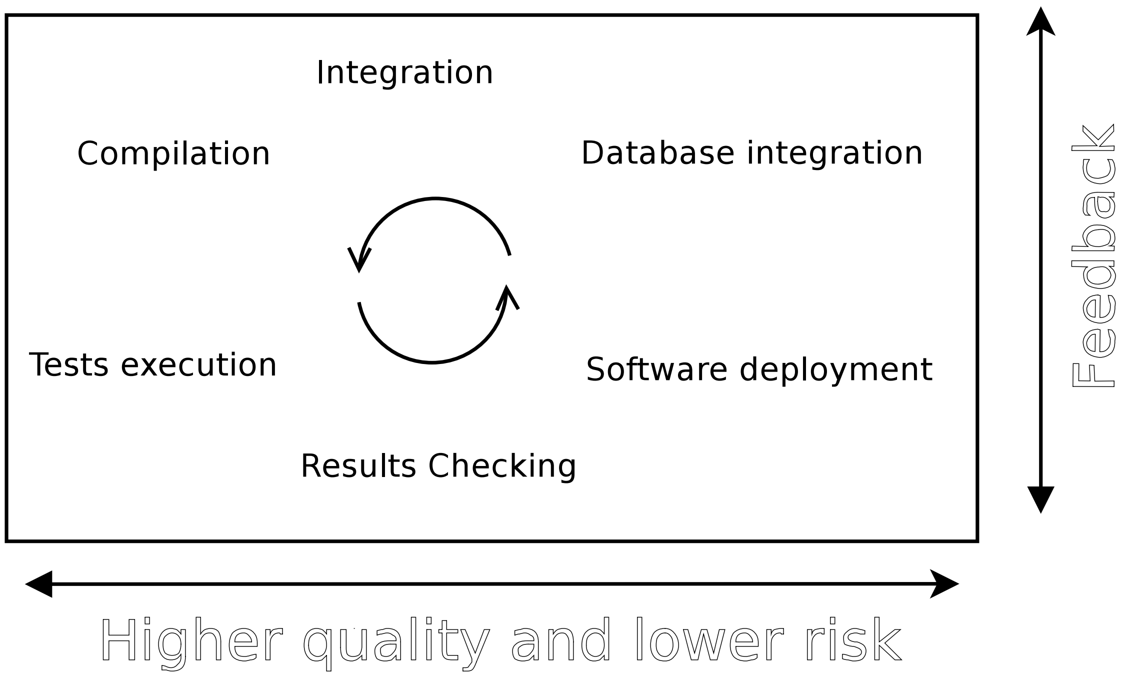

Figure 2.9: Continuous integration

of the tests is necessary. Nevertheless, the developer must choose the right testing scenarios and keep the high standard of tests.

The most widely used continuous integration and continuous deployment server is called *Jenkins* [51].

The development of the Jenkins project was started by Koshuke Kawaguchi in 2009, who worked for the Sun Microsystems. Nowadays the Jenkins project keeps more than 70% of the continuous integration market, which makes it by far the most widely used tool. In 2009 the Oracle bought the Syn Microsystems. This step led to conflicts between the founders of the project and the developers from the Oracle company. The Oracle developers fought for a longer development cycle with heavier testing, while the founders led by Koshuke Kawaguchi stuck to the concept of the open source together with flexibility and swift development. In 2011 the project was renamed. The Hudson was renamed to the Jenkins and it was separated from the Oracle. Majority of users are faithful to the newly developed project. However, the Oracle still continues with its own development.

The Jenkins is an open source continuous integration server. It is implemented in the Java language. It has a very simple interface, which can be easily customised by a large number of plugins. The plugins can be divided into several categories:

- *Version control system plugins* - plugins that provide interface to the most common *Version Control Systems* (VCS),
- *Executor plugins* - plugins that allow execution of certain scripts, such as Python,
- *Interaction plugins* - plugins that allow an interconnection between jobs, for example Join plugin,
- *Metrics and visualisation plugins* - this group of plugins allows a visualisation and

provides support for various kinds of results.

One of the biggest advantages of the Jenkins project is the speed of development. There are updates and bug fixes available every week. There is also a more stable version that is released three times a year. This version contains only packages and bug fixes that are considered stable.

There is no other tool that can match Jenkins in the number of installations or available plugins. However, the swift development and high number of plugins has its drawbacks. Very often the plugins are not compatible and it is not uncommon that development of certain plugins is dropped in favour of a plugin with similar functionality. There are usually several plugins that can bring similar functionality and it can be quite difficult to find one's bearings in them.

The continuous integration server can be understood as a system that maps the set of jobs on the set of nodes. To make this clear a bit, the continuous integration server contains a set of defined jobs that should be executed. At the same time, it keeps the nodes. Each node possesses a defined number of executors. One executor can run one job at a time.

2.7.1 Node control system

The node in the Jenkins environment is controlled by the master computer. The master is the computer where the Jenkins server is installed. The newer versions also support multi-master settings. The master node keeps track of all jobs which are currently configured inside the Jenkins installation and it also keeps track of all the nodes. It controls the execution of all the jobs and it sends the jobs to the slave nodes and, once the job is finished, it tracks the results. It can be seen as the master-slave architecture.

The nodes are controlled by the Java application called `slave`. The slave agent is executed at the node and works over TCP/IP. Once the node is configured it is shown in the Jenkins server as up and running. The jobs can be configured to run on a specific node or group of nodes. If the user chooses a group, one of the nodes is selected by a ballot. The multi-configuration jobs keep the matrix with nodes and this way it is possible to create a single job that will be executed across all supported systems.

Apart from the parallel run of a single job on multiple nodes, there is also a possibility to run multiple jobs on one node at the same time. Each job has its own workspace directory, which means that the jobs are independent. The number of parallel jobs can be set by the number of executors. By default each node has one executor.

2.7.2 Jenkins as a build environment

The Jenkins is nowadays widely used as a tool which performs nightly builds and tests.

In Fig. 2.10 I depict a build pipeline. The whole process starts with the building of tools [67]. The Jenkins environment provides special types of jobs, for example a `maven` job for the Java projects. One of the biggest advantages of the Jenkins is the selection of the nodes where the job will be performed.

Let me introduce the most important steps of the build. The build is a job in the Jenkins that is configured in an appropriate way. I use two kinds of job for the build, the `multi-configuration` job and the `maven job`. The jobs differ just in the execution step, otherwise they are very similar.

The first important feature that can be configured is the job security. The job can be configured in a way that other users can just watch it or control it, etc. There are several

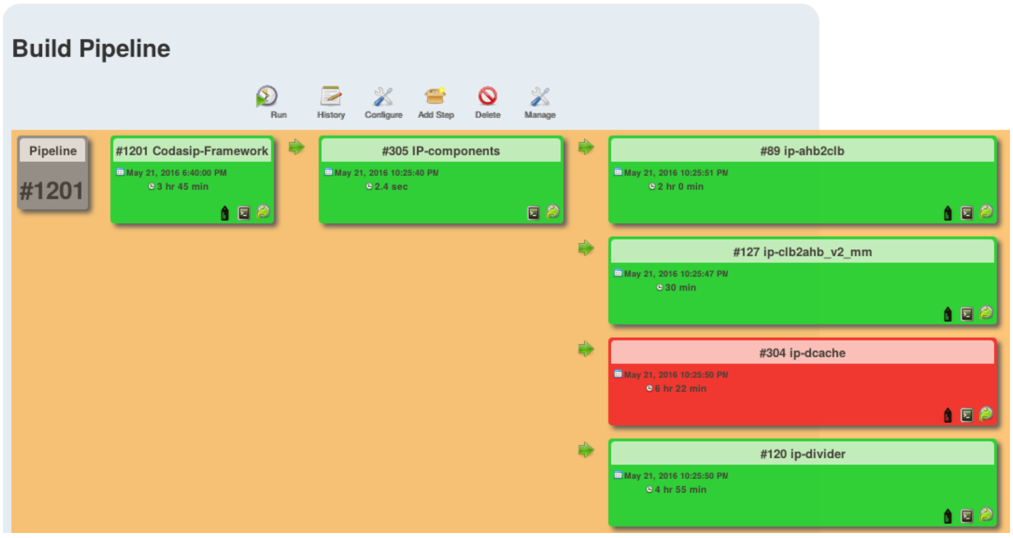

Figure 2.10: The build pipeline

plugins that modify the basic functionality of the Jenkins in this area. One of them is *Matrix Authorization Strategy Plugin* [52] and also very popular is a combination of the *Folders Plugin* [49] and the *Role Strategy plugin* [53]. All variants have advantages and disadvantages, but for a larger company a combination of folders and a role based access is more useful, although it is harder to configure it properly.

A user can also set the names of the jobs that will be able to copy the artefacts in the configuration if the job stores any. Moreover, the job parameters can also be configured. In the Jenkins, there are basic kinds of parameters, such as boolean, string, text and new kinds are added by the various plugins. I can mention, for example, the `Build selector for Copy Artefacts` or the `Choice Parameter`. It is also possible to use global parameters and parameters can be also passed from an upstream job. In this case, the local parameters are overwritten.

Another extremely important part in the job configuration is the *Source Code Management*. All version control systems can be added into the Jenkins environment via plugins. Let us have a closer look at the git possibilities in Jenkins. There are plugins for integration with git [41], such as Gitlab, Github and also GitBucket. There is also a possibility to utilize the git change log. However, the majority of functionality suitable for the majority of clients is in the *Git plugin* [50]. The plugins offer possibilites for advanced checkout and clone behaviour, have cleaning and polling routines and also offer possibility to checkout into a specific directory. However, I have experienced on Windows systems, that it has limitations when checking the various branches of the same repository. The *git* executable and permissions are configured in the global Jenkins configuration.

Then there are the sections *Build Triggers* and *Build Environment*. In these sections, the user can configure a periodical build. This is useful especially for nightly builds and tests. Also the polling can be configured there as well as other actions, such as execution. What is extremely useful is the build abortion. There are several possibilities, such as the *absolute timeout* or the *conditional timeout*. Also the environment variables can be set for injection into the job.

A very important part in the multi-configuration project is the *Configuration Matrix*.

The most frequently used axis is the one containing nodes. The user can define what slaves will the build be performed on. It is possible to choose *Labels* or *Individual nodes.* Also another axis can be added, such as an axis based on a version of the Java language.

All the above mentioned sections can be considered a configuration. After these steps comes the build. The build is divided into the *Build* and the *Post-build actions.*

In the *Build* section, the user can configure an execution or a conditional step. From my experience, it is better to configure the execution and do the conditional steps inside the scripts. There is also a possibility of executing other projects before the execution starts. The kind of the offered executors is affected by the installed plugins.

The last part is called the *Post-build actions.* The possibilities offered here are wider than the ones in the *Buid Step.* It is possible to execute some clean up procedures and also wait for other projects until they finish the build. Very often, the job archives some artefacts and they can also be configured in this step, as well as the trigger of other jobs. Another wide area of the post build actions is the publication of test results. The Jenkins offers a support for all major formats, such as *JUnit* and others. There is also the *Editable Email* configuration that enables sending emails with various features.

The job is stored in the xml format in the Jenkins. The extensions just bring the new marks into the existing jobs.

Using the functionality provided by Jenkins, it is simple to automate the building process. However, for large companies, it can be quite difficult to maintain all the jobs by hand.

Especially in cases when the development of new features that need to be tested, it can be difficult to create new jobs that are needed for testing manually. So I will have a look at the possibilities of the job generation.

2.7.3 Current possibilities of the job generation

Let us have a look at the current development in the field of job generation. I can distinguish between two types of solutions. There are tools in the Jenkins that were designed for this purpose and then there are several works that try to deal with the problem of job generation outside of the Jenkins environment.

First I will have a look at the solutions inside the Jenkins. One of them is the *Template plugin* [3]. Via the template project plugin, the user can set up a template project, containing the settings the user wants to share. Is is possible to set, for example, VCS repositories that are common for the jobs or a script that should be executed and so on. Then it is possible to create another project from the created template inside the Jenkins. So the generation has to be performed manually by using the template several times. Therefore, the possibilities of the automation are quite limited.

Another possibility provided by the Jenkins server itself is the *Job generator plugin* [2]. This plugin is based on the template, which is the job itself and the parameters, which can be global or local. This plugin is very powerful in combination with other plugins, such as plugin for the conditional resolution. However, it shows limitations in the form of what types of jobs can be generated and it cannot use time triggers. Moreover, it is very difficult to generate more complex jobs. The hierarchy and conditions can become very complex and the whole process is quite error prone. I also did not find a way how to set the desired nodes in the multi-configuration project.

The most powerful solution from the Jenkins itself is the *DSL plugin* [1]. The dsl plugin offers the possibility of definition of the job, which will serve as a template. From this template the Jenkins is able to generate other jobs. This is done via a special build step called `Process Job DSLs`. The build step executes the script in the Groovy language. This solution allows the user to perform basically any customization over the template. The Groovy language is very powerful. On the other hand, this solution is still within the Jenkins environment and can be affected by other plugins, which can cause problems. Moreover, the Groovy language is not very common and may require complicated settings.

Now I will introduce several approaches that try to deal with job generation outside the Jenkins environment. Interesting ideas were proposed in the article at the Jenkins User Conference [61]. The article deals with the automation of testing in the area of robotics. The author uses combination of various Jenkins plugins for packaging and a static analysis. Nevertheless, the process of the building and testing is very complicated and hardly maintainable. The author of the article proposes the use of the Domain Specific Language (DSL) for the specification of information and then generation of the Jenkins jobs. It seems that the author just uses the Jenkins for the building. However, the system seems to be slow and problematic as far as the synchronisation of the jobs is concerned. Also there are problems with the graphical side of the solution.

Quite interesting ideas connected to the job generation are described in the Shaw article [89]. The article also introduces the possibility of job generation from templates and usage of the Jenkins command line interface. Nevertheless, the article does not provide any examples of the templates or the scheme how the system works.

Above I have introduced several possibilities in the area of job generation. None of the approaches that were mentioned suit my needs. In our project, I need to generate all kinds of jobs, as it is crucial to test various aspects of the newly developed core. These aspects include the tests of various features that can be tied to very specific kinds of jobs. The approach mentioned in [61] seems to be interesting. For use in our research project it appears to be too cumbersome. A lightweight solution with the command line interface would suit my needs better.

2.8 Disadvantages of the current state

The area of the hardware software codesign is under rapid development and many solutions used in this field are unique. Currently, there is no suitable solution that would even remotely meet my requirements for a complex testing system for the automatically generated compiler.

In the case of test-suites, I need to add the test selection mechanism because compilers have various restrictions. This very often leads to a large number of false negative errors. The designed compiler very often should not be able to compile certain tests. The test-suites are very often hard to scale. By scaling in this context, I mean the ability of the test-suite to execute a set of tests for a given platform that has certain properties. A very common test case is to execute a set of texts that are suitable only for 16-bit architectures.

Also the different types of reports cause problems during the evaluation of tests. Each test-suite, except for the *gcc* one, has its own system of reporting errors and storing tests. In some cases, there is a possibility of storing temporary files but sometimes it is complicated.

In the case of random generators, the usage is also complicated, as they usually provide test cases, which are very difficult to debug. I came across several solutions in this area, and I did not discover any way how to instrument the generators to focus on certain constructions. Also the development in this area is slowly dying out. Overall I can say that they are not meant for use in the area of embedded systems.

Due to the situation described above, companies in this field have to develop their own solutions. They usually do not publish the testing methods they have developed and use them internally. In the thesis, I will try to sketch the practices that help to improve testing of an automatically generated compiler and also give comparison where possible.

As far as compilers themselves are concerned, we saw that a modern compiler consists of many interconnected parts, which are very error prone. Thanks to the study of the compiler internals, I now have a very good insight into compilers and can focus on testing. From the research it is clear that the testing approach must be very easy to modify because the release cycle of the LLVM is quite swift and I will very often need to test various versions of the compiler in the shortest possible time.

To perform the testing, I will use a continuous integration server. I have chosen the Jenkins environment as it suites my needs the best. It provides the ability for various builds. There are also other solutions that provide better functionality in certain areas, but in this case the whole is more than a sum of its parts. No other environment provides all the desired functionality.

The user must have the support of the C compiler library. Without it, the number of constructions that can be tested is very limited. So far, there has been no automatic or semi-automatic process of porting of the library that would be suitable for the area of hardware software codesign.

Nevertheless, the techniques themselves are not sufficient enough. There also has to be a correct release cycle that has to be adhered to. Because the technique can be exceptional but when we do not have enough time for practising the technique, it is useless. In companies there usually is not enough time for the testing of the software. This thesis will also take into account the model of the release cycle, sketch what the key milestones in keeping the quality of the compiler are. I believe that only by combining the proper techniques with the correct model of the life cycle, we can reach the desired results.

Chapter 3

Lissom project

In this section, I will describe the Lissom research project [63], which creates the background for the testing methods that are described in this thesis. The Lissom project started in 2004 and is located at the Brno University of Technology, Faculty of Information Technology, Czech Republic.

The Lissom project has two main areas of interest. The first one is the ADL called CodAL, for the ASIP description. The description of the language can be found in detail here [70].

The second scope of the project is the generation of the full toolchain from the description in the ADL CodAL language. The generated toolchain contains the C compiler, assembler, linker, disassembler, two types of simulators (instruction and cycle accurate), the debugger and a few other tools. As the language is designed for description of the ASIP, the scale of processors that can be described without modifications made to the language is vast.

However, there is also another way how to utilise such a language. It is the use for description of architectures that already exist. Therefore, I can model architectures, such as MIPS [98], ARM [10], RISC-V [86] and many others, in the CodAL language. The generated toolchain or just separate tools can be used as a replacement of the existing tools when they are not in a good shape. This fact offers large possibilities when the core is upgraded and a new toolchain is needed. Also for certain cores, some of the tools might be missing and by designing the given architecture in the ADL, the missing tool can be easily generated.

3.1 CodAL Language

The CodAL language falls into the category of mixed ADLs. This means that the language is able to describe the architectural information needed for the generation of the C compiler and, at the same time, to provide information about micro-architecture, which is needed for the generation of the hardware.

The CodAL language is special for the fact that the description of the core is created in two levels of abstraction.

- instruction accurate,
- cycle accurate.

The first one, the *instruction accurate*, is on a higher level of abstraction. This description is very simple and it is written in a C-like code. It describes the instructions. The addition of the instructions is very straightforward and for an experienced user, it takes only several minutes to create the first version of the core with few instructions for which the basic tools, such as an assembler and simulator, can be created. The designer can fully focus on the instruction set without considering the complicated micro-architecture. From this level of description, the user is also able to generate the C compiler and the profiler.

The *cycle accurate* model is more complicated. On this level, the micro-architecture is described. Things, such as pipeline, hazards, etc. must be taken into account. This description is taken as a base for the synthesis. This level of abstraction gives the user a possibility to generate the description in the hardware description language, the functional verification environment, the simulator, the assembler and the profiler.

There is a large number of files that are common for both descriptions and these files are shared between the descriptions. There might be several equivalent descriptions on the cycle accurate level that correspond with one instruction accurate model. This is logical, as the instruction set must be the same, but there might be several hardware variants that are optimized for the speed or power consumption.

The system of two descriptions is also suitable for the automatic equivalence checking. Nowadays, advanced verification techniques are used for this task. The instruction accurate description is used as the *golden model* and compared to the cycle accurate one. The simulator is generated from the golden model and the hardware description is generated from the cycle accurate model. After the execution, the content of the memory and registers is compared. The verification environment is in detail described in the paper on functional verification [91] and in the thesis [90]. I will not give a detailed description of the cycle and instruction accurate models in the thesis as it is not needed.

3.2 Toolchain

As I have mentioned before, the automatic generation of the full toolchain is one of the two main tasks of the Lissom project. The generated toolchain contains all tools known from other toolchains but it also contains specific tools.

The toolchain that is described below creates an entry point into the testing of the compiler. The generation itself is very often also a part of the testing. Moreover, the toolchain stands as a prerequisite for the tests of the compiler.

All the tools are generated from the description in the CodAL language. At the beginning, the model in the CodAL language is validated and compiled. The result of the compilation is the XML representation of the model. The XML format was chosen intentionally as there are other tools that use this form and there is also a large number of generators and parsers working over the XML.

Once the XML is created, there are two tools working over it. These tools are the toolchain generator, called also *toolsgen*, and the semantics extractor or *semextr*. This approach is depicted in Fig. 3.1. Please note that the scheme was simplified and does not contain all the generated tools.

The toolchain generator produces tools, such as the simulator, the assembler, the debugger and so on. The tools that are generated by the toolchain generator consist of two types of files. Both types of files are compiled and linked together.

1. The files that are *platform independent* are the same for all architectures. Into this category fall user interfaces with parsers of the command line arguments, or in the case of aprofiler, the generation of the graphical output.

2. The files that are automatically generated, such files contain *platform dependent* information. Into this category fall the instruction decoders in the simulators or assembler printer in the C compiler.

The second tool is the *semantics extractor*. This tool was thoroughly described in the dissertation thesis [47]. The semantics extractor is the prerequisite for the compiler generation and also decompiler that is described in the thesis [54].

The semantics extractor is used for the extraction of the semantics, assembler syntax and binary encoding of each instruction. It uses sections `assembler, binary and semantics` [47]. The information extracted from the section `semantics` is transformed into an LLVM-IR like code that describes the behaviour. The information from the sections `assembler and binary` is used to get the assembler syntax and binary encoding. Below there is an example of the extracted semantics for the *nop* instruction.

```
instr i_nop_halt__opc8_nop__, ok (0),
{ },
nop();
,
"nop",
0b0000000000000000 ,
"",
"el:i_nop_halt(el:opc8_nop)",
"class_basic"
,
{{0}}
```

The extraction of the semantics is possible only from the instruction accurate model. The extraction from the cycle accurate model is not supported. The information for the semantics extractor is contained in the suitable form only in the instruction accurate model. Therefore, if the user wants to get the toolchain together with the hardware it is necessary to create instruction accurate as well as the cycle accurate model.

Once the file with the extracted semantics is created, it is used by a tool called *back-end generator*. This tool creates the only platform dependent part of the C-compiler, the backend. The rest of the compiler, the frontend in this case the Clang and middleend, the optimizer are platform independent. The backend part of the compiler uses the information from the semantics extraction for pattern based matching for the most suitable instruction.

Also other tools can be generated. One of the basic tools is the assembler. In this case, the assembler is not special in any way. It takes the input in the assembly language of the given ASIP and produces the output in a form that is suitable for the standard linker *ld*. Nevertheless, the output format of the assembler, which is the input of the linker, contains non-standard enhancements.

Also some of the tools can be generated from the cycle, as well as the instruction accurate, model and this fact is not reflected in the scheme.

The tool that is inverse to the assembler is called the *disassembler*. The disassembler is used for the translation of the binary file back to the assembler representation. It is used when no description in the higher programming language is available. The code produced by the disassembler is more difficult to read than any code produced by the programmer, even in cases when the original code was also produced in assembler. The tool can be useful for the debugging.

There is also an application which is called the *randomgen*. This tool works over the extracted semantics, but over a different kind. There are several types of extracted semantics

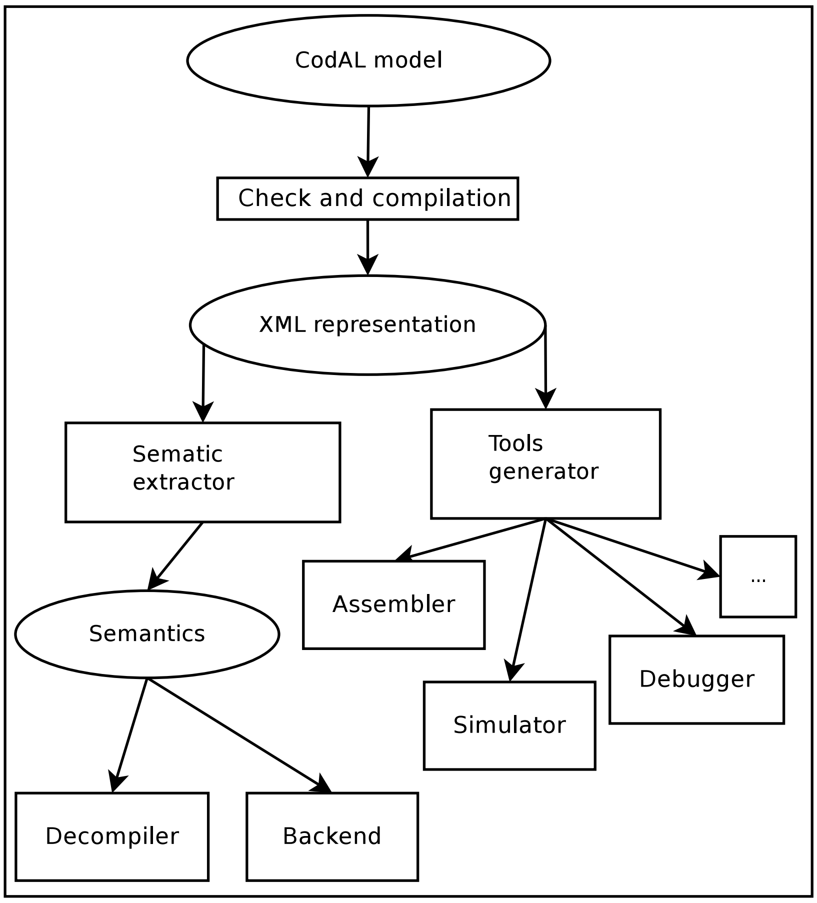

Figure 3.1: Scheme of the toolchain generation

that are used for various generators. A random generator is able to generate valid random assembler applications. These applications are typically used for the functional verification. The programs can be simulated but it is not guaranteed that the exit code of the simulator will be correct. However, the content of the memory and registers must be identical, despite the used tool. The user can specify the number of lines of the generated programs. It is also possible to set how many applications should be generated.

Three types of simulators can be generated, the instruction accurate one, the cycle accurate one and the QEMU simulator [85]. Each of the simulators has a different area of use. They largely differ in speed. The fastest of them is the QEMU simulator. The instruction and cycle simulators are generated from corresponding models. The QEMU simulator is the fast instruction accurate simulator and needs information from the instruction accurate model.

The QEMU is in fact a virtual platform. In our simulator we use just a fraction of the QEMU functionality. The QEMU creates a virtual machine in the computer that emulates the CPU.

The slowest of the simulators is the *cycle accurate* one. However, the cycle accurate simulator is as close to the hardware as a simulator can be and, therefore, it can provide information that cannot be obtained from any other simulator. The use of the simulators is given by the characteristics. The *QEMU simulator* and the *instruction accurate* simulators are used for benchmarking and larger software programs. In the cycle accurate simulator, it is possible to get the number of cycles and other characteristics that are needed when the design is going to be put into real hardware.

The instruction and cycle accurate simulators can be configured to generate profiling information. The QEMU simulator does not have this ability. There are two levels of profiling. The higher level of profiling information can be gathered from the instruction and cycle accurate simulators. When I want to get low level profiling information, I need to use the cycle accurate simulator. The information is saved into a file and can be processed by the profiler.

All the tools I have described above can be generated with various optimizations and the verbosity can be set from 0 to 3. The zero level is the silent mode. When the user chooses level 3, it prints the maximum amount of information from generation and runs of the specific tool.

Profiler is a tool that is used for the dynamic program analysis. It gathers information about the memory, complexity, usage of the particular instructions and the frequency and duration of function calls. The profiling information is used for the program optimization. In our case, the profiler is also generated and can be generated either from the cycle or an instruction accurate model. The information from the profiler is visualised in the special view of our tool.

The exported libraries can also be part of the toolchain. Currently, it is possible to export *compiler-rt* for the 16 and 32-bit processors. The Newlib library can be exported only for the 32-bit processors because the support for the 16-bit is very limited.

The structure of the exported toolchain is in the example below.

```
toolchain
|-- bin
|   |-- urisc-assembler
|   |-- urisc-llc
|   |-- urisc-clang
|   |-- urisc-isimulator
|   +-- ...
|-- lib
|   |-- clang
|   |   +--include
|   |       |-- limits.h
|   |       |-- stdarg.h
|   |       +-- ...
|   |-- libcomp.a
|-- newlib
|   |-- lib
|   |   |-- crt0.o
|   |   |-- libc.a
|   |   |-- libnosys.a
|   |   +-- ...
|   +-- include
|-- contrib
    |-- libc.so
    |-- libdl.so
    |-- libz.so
    +-- ...
```

The structure above is quite clear. All the binaries are in the directory `bin`. The directory `lib` contains the libraries, mainly the compiler-rt, which is called `libcomp.a`, which is optional, it does not have to be a part of the toolchain. The directory `newlib` contains the exported Newlib library, which also does not have to be included. Both directories, `lib` and `newlib`, have the subdirectory `include`, which contains the header files.

The directory `contrib` is also part of the toolchain. In this folder, I keep the files that are needed for the proper functionality of the toolchain. Typically the shared libraries are kept there. The libraries are taken from the system and this step helps to keep the toolchain partly system-independent.

In our project, we have the IDE based on Eclipse. It is basically the client who is able to visualise and launch all the tools. It is a thin client that contains the editors, configurations and many other functionalities. There are editors for all the formats which are supported. This includes the CodAL language, the assembler, C language and several others. There are also various configuration tools and browsers for the help, which are embedded in the IDE.

The environment also contains several perspectives, which help the user in various stages of the development. A perspective is a special kind of a window, which is customised for the display of a certain kind of information. There is the basic perspective used for editing of various files, the debugging perspective and profiling perspective. However, it contains minimum functionality as far as generation is concerned.

As the generation of all the tools is concerned, it is the task for the command line. This division allows the user to work completely without the graphical interface. I can say that the IDE provides the subset of the command line functionality.

Chapter 4

The goal of the thesis

This thesis is focused on finding new attitudes to testing of the compiler and the whole toolchain for the hardware software codesign. The current ways of compiler and toolchain testing are not in many areas tailored to fit the rapid development of the ASIPs. Moreover, I need to keep in mind that this system will also be used in a commercial company so I need to pay special attention to the usability. The attention will be paid mainly to these areas:

1. *Raise the number of tests that can be used for testing.* The number of constructions that can be used for the testing purposes is very limited. Without the standard C library a very limited subset of the C language can be used for testing. The author will try to increase the number of programs that can be used for the testing by choosing a suitable C library and by semi automatic porting on various cores.

2. *Develop a test selection mechanism.* As I test various cores, which are not interchangeable, I need to have a very good test selection mechanism. I need to have a simple form of enabling and disabling either tests or whole dictionaries. This approach should have a high level of automatism. Also the deployment of generator would be welcome.

3. *Accelerate the whole process of the testing.* The testing, especially when I need to test for various operation systems, can be very time consuming. I need a way to speed-up the process of testing. I will examine the process of testing and try to find a way how to pre-build certain parts of the toolchain and, if possible, I will also inspect and speed-up the creation of the nightly builds.

4. *Develop a way of automatic generation of the jobs based on test parameters.* The execution of tests is nowadays performed by continuous integration servers. Nevertheless, because I need fast and flexible management of the jobs, I need to provide an automatism for generation and upload of the new jobs to the continuous integration server.

All the mentioned goals and techniques will be tested and evaluated for several cores, which will vary in size, power consumption and complexity. The experimental results will be taken from these cores where possible. The solution will be used on various operation systems at least on the Windows and Unix platform, so the implementation language will have to be chosen with respect to this condition.

4.1 Solution phases

For the first point, I will try to find a suitable C compiler library that will enable the use of tests which are dependent on the calls of the library functions. I will also try to automatize the porting process as much as possible to keep the porting process for the new core to the minimal extension.

As far as the second point is concerned, I have demonstrated that, while there are various sources of the testing programs, the mechanism that would help with the selection of the tests is missing. I will propose a test selection mechanism that will be lightweight and will allow simple addition of the new test. It should also support the generation in a way that, once the new platform needs to be added, the corresponding files which provide the functionality of selection can be generated.

The third point deals with the speed of testing. From the presentation that was described at the beginning of the thesis, it is clear that developers are interested in deploying the new build several times a day. Hand in hand with build process goes the testing process. I will try to find a way of accelerating the testing process within the continuous integration environment by pre-generating the files that are needed for testing. Via the generation, I should be able to achieve time, space and traffic savings.

The last point is closely related to the continuous integration systems. In the section of the related work, I have discussed the current possibilities of the job generation. It is apparent that this area is not very popular as there is not much happening. Articles that discuss this area are very few. I will introduce the generator of the jobs that will work over a set of templates and will have the ability to generate a wide range of jobs.

Chapter 5

Porting of the C library

The first part, which is needed for automatic compiler testing of processors for embedded systems, is the support of the Newlib library [26], [32],[30]. The variety of programs that can be created without the support of the standard C library is very limited. Therefore, the availability of the library is crucial and its position in the process of testing is unsubstitutable. I have worked on the first version of the Newlib port that will be described here.

5.1 Theory of Porting

The main reason for porting the library on the new platform is the fact that I need to add support for the call of the C functions. To be precise, I want to use the *libc* functions, such as `printf, malloc, free`, etc. in programs that will be used for testing of the compiler. And because I do not possess the development kits for all the platforms, I use simulators instead. Therefore, I must add the new platform into the Newlib library and our simulators must know how to deal with the Newlib library calls. If one does not grant libc library support in the simulated environment, the number of constructions which can be used and tested is very limited. Consider the following simple example written in C:

```
int main(int argc, char **argv)
{
    if(strcmp("alpha","beta")==0)
{ return 1;}
    else
{ return 0;}
}
```

Even this simple program can hardly be executed because it uses the function `strcmp` that is part of the standard C language library. This program cannot be compiled, unless the file of `string.h` is included and a possibly some other header files are included also.

On the contrary, the main aim of the testing process is to cover as wide area as possible and also to try as many different combinations of the function calls as we can. However, this goes against the idea of embedded solutions, which are usually specialised in just one single area. Furthermore, because I focus especially on the embedded systems, I do not even try to cover all the functions provided by the standard C language library, which is in my case the Newlib. In fact I will use and therefore test only those functions that can run under the simulated environment and are useful for the programs that will be executed on the given platform. Moreover, the embedded systems are not designed for the use of the vast number of constructions that the programming languages offer these days. Typically there is just one task, usually quite a complicated task, which is launched repeatedly. However, during the design of the chip it is often unclear what part of the library will be needed, so I will have to port the whole library and reduce the size later if it is necessary. There are certain areas that are more likely to be removed from the library than others, for example:

- *threads* - I assume that in simple programs for embedded systems one will not use threads.
- *locales* - All the locales were removed from the library.
- *inet module* - Even though networking plays an important part in modern embedded systems, in some cases the module can be disabled.
- *files and operations with files* - Certain simple application do not need interface for working with files.

Now I will introduce important parts of the library. Simply said, all that really has to remain from the library are the `sysdeps`. The `sysdeps` are the core of the whole system (how to allocate more memory, etc.), then important modules, such as `stdio`, which takes care of the outputs and inputs, and other modules I wish to preserve. In this case, I wished to preserve the following parts of the Newlib library:

- *stdio* - This is one of the main reasons for porting the library, which is to get in human readable form output from the simulator.
- *module for strings and memory* - In many applications I would like to use functions, such as memcpy, strcpy, strcat, etc.
- *memory functions* - For example `malloc, free, realloc,`
- *abort and exit.*
- *wchar support* - But without the support of different encodings.

Some parts of the library could not be removed because of the dependencies. According to my estimations, nearly 40 percent of the library was disabled or removed, measured by the size of the library.

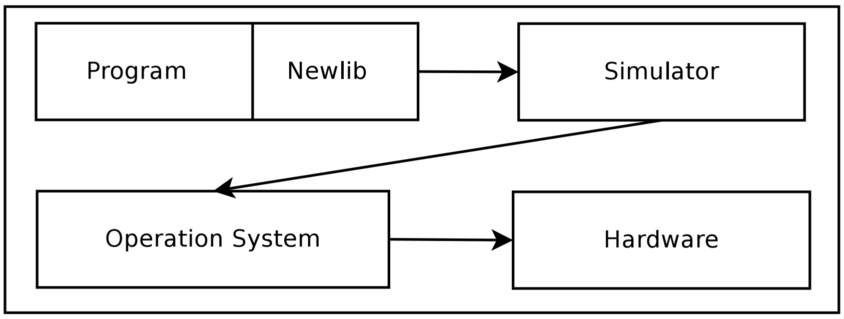

Figure 5.1: Position of the Newlib

There are several ways of building the library and also different methods of using it. There is a possibility of building the *Position Independent Code* PIC. Even though this is an interesting solution, I decided against it. Instead of the PIC, I am going to compile the library into a single object and then link it to the program. The scheme of the position of the Newlib in testing is in Fig. 5.1.

Now let us return to the functions that remain in the library. The functions can be divided into two groups. The first group consists of functions that are completely serviced within the simulated environment. For example, the function `strcmp` falls into this category. This function and its declaration remain unchanged within the simulator if they are written in the C language that does not require any changes. These functions are not tied to a kernel header files, so there is no need to change them.

The second group of functions consists of functions that are translated to the call of system function. The function `printf` can be used as an example of this group of functions. The call of `printf` function can be divided into three phases that are illustrated in the following picture 5.2.

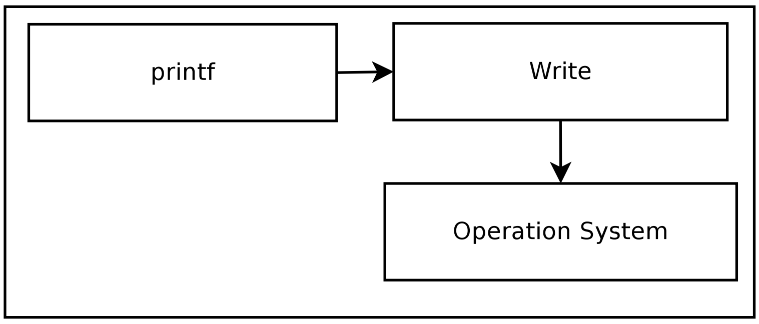

Figure 5.2: Scheme of the printf function call

At the beginning, the call of the `printf` function is translated to the call of a system function, with the highest probability it is going to be the call of the function `write`. Write is the function call, that is serviced by the operation system, and hence is system dependent. But as I want to use the simulator on the UNIX platform, as well as on the Windows systems, I have to get rid of these dependencies. To do so, I will use the special instruction principle.

5.1.1 Use of ported library of UNIX and Windows systems

Before I get to the principle of the special instruction method, I should explain why I need to use this method. The main reason why I should oust the dependencies on the kernel header files is the fact that I must be able to use the library under UNIX systems and also under Windows like operation systems.

As long as I use the library under UNIX systems, everything should be all right. Though, even on UNIX systems there might be differences between the different versions of the header files. But once I use the Windows based system, I cannot use header file functions anymore. It would almost certainly result in a crash of the system.

In our project, I currently support several UNIX distributions as well as the Windows. The use of other operating systems is not considered.

5.1.2 Special instruction principle

The special instruction principle means that I will use an instruction with the *OPeration CODE*, *opcode* that is not used within the instruction set for a special purpose. So far all architectures that were modelled within our research project had several free *opcodes*. It is typical that the instruction sets do not use all operation codes which are provided. But in the case of no free opcode, this method cannot be used. The special instruction principle will be used for ousting the dependencies on the kernel header files.

Functions provided by the operation system are triggered by the *syscall mechanism*. The system calls can be quite easily detected. Each library should have defined the syscall mechanism in a special source file. This syscall mechanism differs, as they usually are platform dependent. So i386 architecture will have a different syscall mechanism from the ARM [10].

I wish to preserve the mechanism. The syscalls will remain in the library, but with different meaning. The file containing syscall will be changed in the following way: at the beginning, the parameters of the syscall will be placed at the given addresses in the memory and I will also define where the syscall return value will be stored. Afterwards, the call of the chosen instruction will be performed. It is also possible to put the parameters into registers, but some platforms have a limited number of registers, therefore, this method could cause problems.

The syscall mechanism is in fact a wrapper of the system call. The call will be passed to the simulator that will do the call and return the result.

5.1.3 Simulators

As was described above, all simulators are generated automatically. At the beginning, the source files are generated by specialized tools. When the generation phase is finished, the simulator is build by the `Makefile` from the automatically generated files and also from the static files. It will be necessary to add the following information into this process:

- Information about which instruction calls the system function.
- The simulator will have to know the convention for storing parameters.
- The simulator will have to recognize which system function is going to be called.
- The simulator will have to perform the call of the correct system function.

The first three points will be solved within the model of an instruction set. The instruction with the opcode that is not used will be declared. The instruction behaviour will be defined in the following way: according to the parameters it will call the given system function. The simulator will have to recognize the system it runs under, and call the correct function. For example, on the UNIX system it will be the function `write` and in the Windows the `WriteFile`. This problem should be solved by the `libc` library of the given platform. The call of a special instruction is demonstrated in Fig. 5.3.

The parameters that were placed at the given position at the simulated memory can remain unchanged. They will be later passed to the specific system call. One important issue is connected to the simulated memory. As I would like to correctly simulate the operations with memory, such as `malloc, realloc`, etc., I need to tell the simulator how much memory it can simulate. This will be done most probably by a special file that will be passed to the linker. This file will contain symbols, which will declare how much memory can be used.

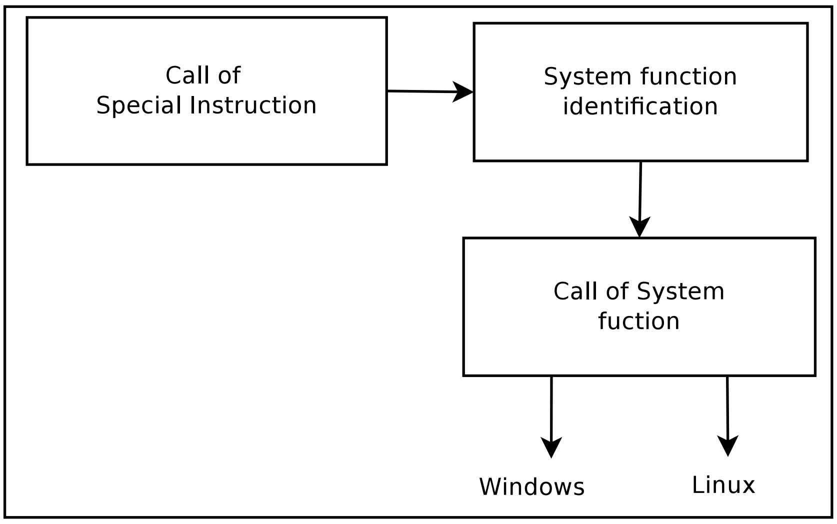

Figure 5.3: Scheme of the system call

5.2 Process of porting

In the early stages of the Lissom project, I had to use other than the automatically generated compiler for the building of the Newlib library. It was mainly because of the fact that the generated compiler was not stable enough. Nevertheless, in the latest releases it is possible to use the generated compiler.

Several issues, which I faced during the process of porting, were closely related to the buildsystem of the library. The library contains a system of makefiles. This system is hierarchical and usually the makefiles from the upper levels are included. If, for example, I would like to compile any test examples that are included in the Newlib, I switch to the given directory and call make. This will call all the makefiles from the above directory. This is very effective because only the makefile in the root directory contains variables defining which the compiler, assembler and linker will be used. On the other hand, it is very difficult to modify this system when I want to build the different parts of the library using different tools.

Currently, I am using the set of our tools containing the archiver, linker, assembler and compiler for the development. The currently used compiler is called the *prefix-clang*. The prefix stands for the given architecture. It is an automatically generated compiler from a model description. The linker and archiver are not generated automatically but were developed within our research project.

I have spent quite a lot of time on tuning our toolchain. Our toolchain is based on tools from the LLVM framework and uses also the GNU binutils. It is consistent with the majority of flags that are accepted by the `gcc`.

The system used for building of the library starts by parsing the configuration file and, according to the content of the file, different macros and variables are set. When doing manual changes to the system used for building, I have basically two possibilities:

- to change the configuration file
- or to do the changes later in the `Makefiles`.

The first possibility is cleaner but the `Makefiles` often check if the option is present in the configuration file and ends with an error when the option is missing. therefore, it is more convenient to do the necessary changes in the `Makefiles`. Thanks to the hierarchical structure, it is in most cases sufficient to do the change in just one place.

In the beginning of this section, I have mentioned the need to link special file containing information how much memory can be used. The file will contain symbols defining the beginning and the end of the memory space that can be used. It will have the following syntax:

```
#file defining memory boundaries
define start 256
define stop 768
```

Given that the numbers are in kB, the simulator can simulate up to `512 kB` of memory. Character # used in the first line denotes comment.

As far as the convention for storing parameters is concerned, I have chosen the following approach: the first parameter says which system function is going to be called. In the Newlib, there is a list of system functions for the UNIX systems and I have added also the names of the functions for Windows systems. The rest of the parameters (2-7) are parameters, which are passed to the function call. The parameters remain unchanged. They are passed to the system function in the exactly same state in which they were saved in the memory before calling the special instruction. The special instruction itself has no parameters. When the instruction is called, all the parameters have to be stored in the memory at the given addresses. The simulator takes the address that is passed as a parameter and has the knowledge of the structure so then it is easy to find the corresponding parameters and perform the call.

5.2.1 First time porting

As for the first time, all the steps were performed manually. In the future, I would like to automatize this process as much as possible. Without doubt I could remove the needless parts of the library automatically. The needless parts would be identified in the configuration file and also the special instruction principle could be highly automatic. If I have a spare instruction, I will choose it and compose it into the simulator. Unfortunately, there are steps that need to be performed manually. For example, I need to provide the runtime for the simulators and the corresponding sections need to be specified in the CodAL file.

The runtime is also one of the files that are written manually in an assembly language. There are also other files written in the assembly language and are, therefore, platform dependent. In the case of the MIPS platform, there were eight files that contained the assembly language. For example, the `syscalls or memcpy` functions are all implemented in the assembly language. In order to minimize the number of files written by hand, I decided to provide as much files written in the portable C as possible. I managed to replace many files by the C implementations. All that has to be provided is the `runtime` and the `syscall` mechanism together with the supporting files.

5.3 Automation of the porting process

By default, the Newlib uses the system of make as was mentioned above. I have put quite a lot of effort into the automation of the whole process [27]. The Newlib library was modified, so it now uses the CMake system. It was divided into two parts that are placed in separate directories. One part is common for all platforms. This part is placed in the directory called the `newlib`. The directories that contain platform dependent files are stored in the directory with the model. This is done in order to have all the platform dependent files in one place in the strictly given directory structure.

Let us have a look at the platform dependent files. Strictly spoken, the directories do not contain only platform dependent files. There are also files that are the same for all the platforms but the division is done on the level of directories and not on the level of the files themselves. The directories that are kept together with the model are the directories `libgloss` and the directory `newlib`, this is the subdirectory of the directory `newlib` mentioned the paragraph above.

While the directory `newlib` contains mainly header files with various settings and definition of the `setjmp.S`, the directory `libgloss` takes care of the syscalls handling. The syscalls are very important for our project because this mechanism allows us to get the information in and out of the simulator. I will focus on the way how to automatize the process of syscalls creation.

There are several ways how to cope with the syscalls porting. After I gathered all the necessary information about what syscalls are necessary for the simulation and tried several ways of implementation, I found out that only a very small part of the syscalls must be written in the assembly language. The rest can be written in the C language and that makes the code platform independent. The Newlib defines 20 syscalls but I need just 6 of them.

Nevertheless, the rest of the syscalls could be implemented in the same way as the six supported ones. The syscalls are defined in the header file and have numbers from 1 to 20. The first six are the supported ones and the rest of the numbers is assigned to the unsupported ones.

For the syscalls themselves, I have defined the structure called `params`. This structure contains the parameters that are needed for each syscall. This structure slightly varies depending on the actual syscall. But it is written in the C, which makes it also platform independent. What is only written in the assembly language and is, therefore, platform dependent is the `PERFORM_SYSCALL` function. In fact it is not a function but a multiple line macro defined in the *inline assembler*. Let us assume that a multiple line macro can have the following form:

```
define PERFORM_SYSCALL(ADDR) \
   __asm__( "REGr1=add REG0,%0" : :"r"(ADDR)); \
   __asm__( "syscall");
```

This macro is not taken from any existing processor. I have defined it just for the model purpose. Now let us have a closer look at the macro itself. This macro takes only one parameter. The `ADDR` parameter is the address of the structure that contains the parameters of the syscall as mentioned above. This address is assigned to the register that is used for passing of the parameters. This register can be specially marked as it is often used for passing of parameters. Then there is the special syscall instruction, in this case it has the name `syscall`. These two lines can be determined from the description of the core performed in the CodAL language. I will propose a way how to create the macro semi-automatically. Consider that the `PERFORM_SYSCALL` macro itself is a template. The necessary information can be filled into this generic template before the compilation time of the library. First let us have a look at the syscall instruction. I simply scan the model for the instruction that bears this name. If the instruction is not found, I search the model for the construction in the following form: When this construction is found, I use this instruction in the second line of the multiple line macro. Please note that in this case, the instruction does not take any parameters. If this instruction was parameterized, I would determine the parameters from the syntax. Nevertheless, this instruction does not have to be found. In such a case, the template would be incomplete and an error should be reported. The process is shown in Fig. 5.4.

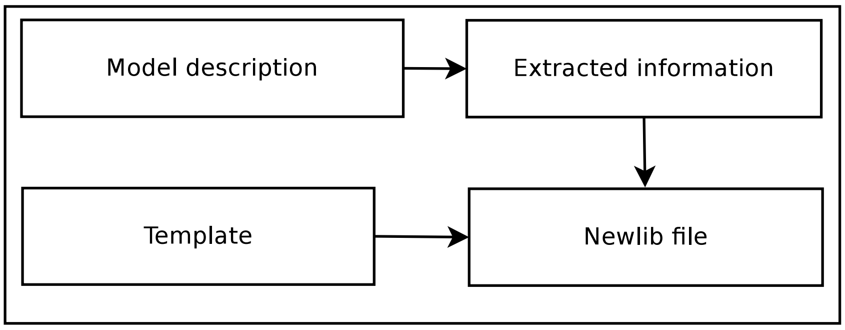

Figure 5.4: Scheme of Newlib file generation

As far as the first line of the macro is concerned, I need to assure that in the register, which is used for passing the parameters, I assign the address of the structure with the parameters. So I search the model for the instruction `add` or instruction with similar functionality. In the syntax section of the instruction, I find the actual form. Then I find the register for passing parameters in the model that also bears special description. From these parts of the information, I should be able to put together the first line of the macro. This approach works for standard architectures. But there may occur architectures for which there might arise difficulties. The Newlib library, in the current version, supports only 32-bit architectures.

5.4 Experimental results and contribution

For having a comparison with commercial compilers, I tested the automatically generated compiler with the commercial Perennial test-suite. The results described here were gained from the generated *MIPS* and *Codasip uRISC* compiler. The testing was performed on a complete toolchain. The tests were compiled by the generated compiler and afterwards executed the tests on the simulator which was also automatically generated by the tools from our project. I have only a part of the Perennial test-suite. I used only tests that examine the core of the compiler. I excluded some of the tests that cannot be compiled due to the header files dependencies, which I do not support. The tests in the test-suite are divided into groups according to the chapter of the standard that is tested. I use tests for the clauses 5 and 6. I have mainly tests for the standard C90 and several tests for C99 standard. The results are summed up in Table 5.1.

In Table 5.2, I present the testing results with and without the presence of the C library. It is apparent that not only the number of tests is lower without the library but also the number of failing tests is very small. The presence of the library provides a better opportunity for debugging of the code and triggers more errors.

The solution also brings a higher level of automation into the testing of the automatically generated compiler. I have introduced methods that simplify the porting of the library to

Core	*Tests without C library*	*Tests with C library*
MIPS	797	1680
Codasip uRISC	804	1688

Table 5.1. Comparison of number of tests.

Core	*Failing Tests without C library*	*Failing Tests with C library*
MIPS	2	19
Codasip uRISC	0	8

Table 5.2. Comparison of failing tests.

the newly developed cores. The porting of several files is no longer needed. Now I only need to write a few files in the assembler language, which is far less time consuming. This is demonstrated in Table 5.3.

Core	*Number of files that must be ported*
MIPS with automation	2
Manual porting without automation support	6

Table 5.3. Comparison of number of ported files

I have chosen the comparison based on a number of files. I could also compare porting times. But the porting time depends on the experience of the developer, complexity of the model and many other things. Nevertheless, the time needed for porting has been shortened from days to hours.

Amongst the biggest contribution I can place the following things:

- *enlargement of the number of tests* - Without the support of the C library, it is possible to test only a very limited set of tests, in my case the number of tests was increased three times.
- *speed-up of porting* - The library was rewritten in a way that it enables far faster porting for new cores, the number of codes which have to be written by hand has been significantly reduced.
- *higher level of automation* - The code that is common for the majority of the cores was introduced, as well as additional scripts for build automation and creation of the library, providing a higher level of automation than before.
- *larger number of failing tests* - It is often very difficult to trigger bugs without the support of the library, so it enables better test coverage and triggers a larger amount of errors that help to keep the compiler in a good shape.

The porting of the Newlib library and topics connected to the porting were published in the articles [26], [32],[30]. The articles describe the process of porting and its automation together with the results.

Chapter 6

Tests selection

As was mentioned in the section which discussed the test-suites, one of the weakest points, which does not suit my needs, is the test selection mechanism. I have decided to create a test selection mechanism that suits the needs of the testing system for the hardware software codesign [30]. It will form the content of the following chapter.

6.1 Test selection scheme

The test selection scheme that would be suitable for use in our project must fulfil several criteria. First of all, it must be independent of the source of the test, so it will be applicable for as large a number of tests as possible. It also must be robust enough and lightweight at the same time, so it should be simple to modify the tests I already have and addition of new tests must not be difficult. It should not only work for tests from the regression test-suites, but should also be applicable to tests from random generators.

6.1.1 Test selection phase

As I have a large amount of tests from different sources, I need a universal approach that will define which tests are suitable for compilation and execution on the given platform.

I have created a system of files, which restricts the number of tests that can be compiled on the given platform, based on the libraries that are available. The libraries are just one of the test selection criteria. Other characteristics are also taken into account, for example, the size of the registers or the size of the stack.

Currently supported features which can be used for the test or directory selection are:

- *architecture* - Certain tests or directories can be disabled for the given architecture.
- *libraries* - Tests can be disabled if a certain library is not present.
- *bit width* - Test selection according to the bit width.
- *level of description* - Often some tests, containing system calls, cannot be used for a cycle accurate model.
- *purpose of compilation* - Some directories are disabled, for example, for functional verification.

The naming convention for the files, which are used for the test selection, is very simple. The file bears the same name as the test does but it has the suffix .x, instead of .c or any other. The system is a hierarchical one. It is possible to have a hierarchy because I support nesting of the directories and I keep the .x files not just for the tests, but also for the directories. In the case of directory, the selection file has the same name as the directory with the .x suffix.

These files possess as minimal functionality as possible. I try to keep their size minimal. The typical functionality of the file is that, based on the value of the flags, the test is excluded from testing. I should say that implicitly all the directories and all the tests are selected for testing. So, if I want to exclude the tests, or whole directories from testing, I have to indicate this.

As the size of the files is kept minimal, the functionality and flag settings must be done in another place. This functionality is kept in the main testing module. The functions that check the current state of the flags and control what libraries are accessible for the linking to the given platform are declared here. The centralization has a purely practical base in this case. The typical usage of the .x files is that I disable testing of the whole directories according to the libraries that are accessible. The .x files can also bear other functionality. It is possible, for example, to set different variables. I can specify flags that should be added to the compilation or add some files to the linker as in the following example.

```
if [ "$C_LIB" == "0" ]; then
     FILE_DEPS+=crt0.o
fi
```

On the level of files, I most often use the .x files for filtering the tests that depend on compiler-rt library for the given platform. The compiler-rt library provides software implementation of the float and double operations. Usually only a few tests in the given directory depend on compiler-rt and the dependence does not have to be the same for all platforms, the best solution is to condition the test execution by the platform and compiler-rt presence. This is demonstrated in the following example.

```
is_arch "mips_basic" $1
      if [ "$?" == "0" ]; then
        if [ "$RUNTIME_LIB" == "0" ]; then
           RUN_TEST=0
       fi
     fi
```

The biggest advantage of this approach, and also the main reason for introduction of this system, is its universality. I deploy the tests from the llvm test-suite [65], gcc test-suite[40], Mibench [72] set of tests and I also have tests that were created within our project, and I have also generated tests. The system of the .x files can be used for all these sources, as long as I use just the tests without the testing infrastructure that is provided in several cases. The only set of tests, which I tried to use together with the infrastructure that is provided together with the tests, is the Perennial test-suite [80]. After several iterations, I have also started to use the Perrenial tests with my infrastructure for the tests execution.

6.1.2 Test compilation and execution

The compilation of tests is performed in the central module. As I have the system of the .x files, I enter only those directories that I know are suitable for testing on the given platform.

So, before I enter a directory with tests, I check the `.x` file for the given source and consult the restrictions that are defined by the `.x` file and set all the variables denoted by the file.

If the directory is feasible for testing, I cycle through the tests in the order denoted by the test list. The `.x` file is always checked first, and if nothing blocks the procedure of testing, the test is compiled. The presence of the `.x` files is not compulsory. As mentioned above, the default setting is to cycle through all the directories and execute all the tests. However, if the file is present, it will be checked. When the restrictions are not met, the file is skipped. The whole process is sketched in Fig. 6.1

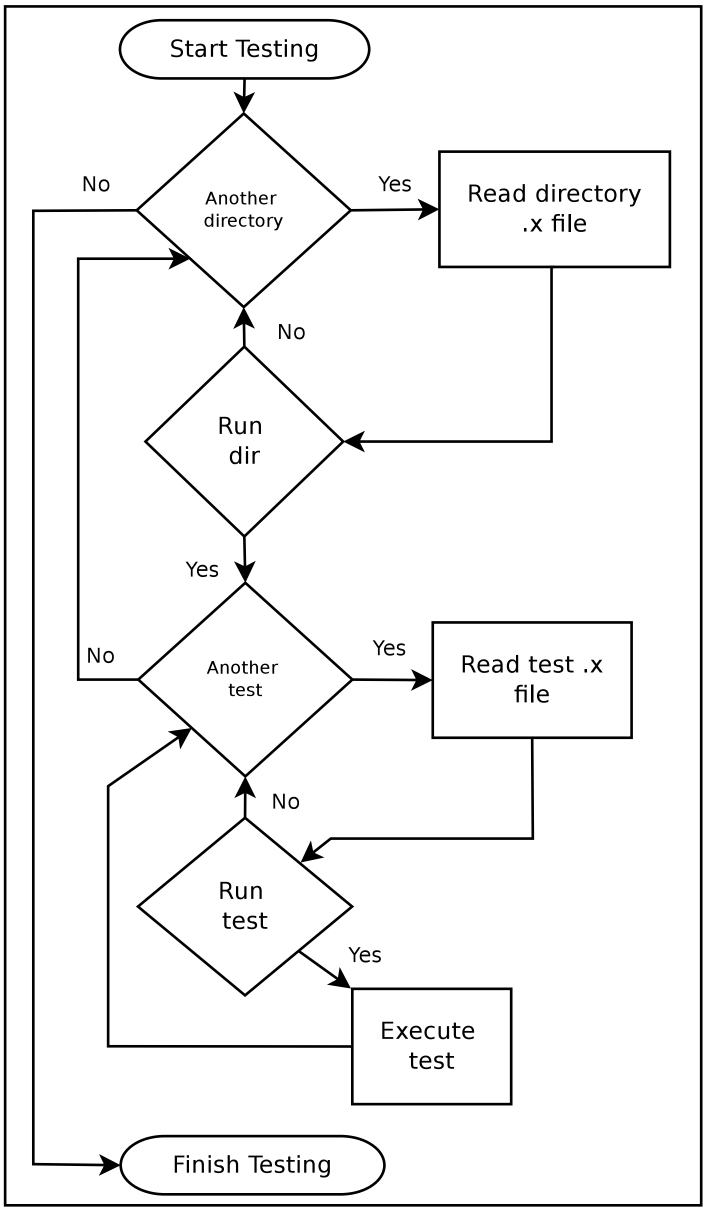

Figure 6.1. Scheme of the .x files invocation

Should there be any problems during the test compilation, they are logged. I log the standard output as well as the error output. I keep a list of tests that were not compiled successfully together with the output of the compiler. The logs are kept for every platform that is tested to avoid overwriting. It is also possible to create a unique log not just for each platform but for every run of the testing system. These logs could be, in the future, stored in the database to keep precise testing history.

6.1.3 Logging information and test evaluation

The test evaluation is kept decentralized. Because I deploy tests from different sources, I need to keep the scripts that provide the test evaluation together with the tests. Some tests are evaluated on the basis of the exit code, but there are tests that produce, for example, the text output and I have to compare the output with referential values. In these cases, the `Newlib` library is used.

The decentralization in this case means that I keep for every directory a shell script that takes care of the test execution and evaluation. Nevertheless, I found out that this system is quite hard to maintain. It seems that the majority of tests is executed and evaluated in the exactly same way. So it makes sense to have one central point of the execution and evaluation and has special scripts just for the scenarios that are not common.

As in the case of test compilation, I keep detailed logging information. I keep the output of the simulator and after the test evaluation I put it into the list of passed tests or failed tests according to the result of the evaluation. The logs are created for every tested platform and can bear the time reference. Below there is an example of an error log.

```
--- vprintf-1.c
simulation error
optimization: 2
return value: 1

--- stdout:
r[3] = 65092 [0xfe44]
r[31] = 13512 [0x34c8]
r[0] = 327832 [0x50098]
warning(0): top_level.mips_basic@498: Unknown instruction

--- stderr:
Instruction decoding failed. Use simulator in debug mode with
   debugger for more details.

--- expected output:
hellohellohello
hello
aahellohellohello
hello
aaxxhello
hello
0
0
exit 0
```

We can see that the test exited with a non zero exit code. The test was meant to print text on the standard output, but this did not happen. Instead of printing, the instruction decoding failed. The log also contains information about values that were in certain registers, what the return value was, and optimization of the test. In the log files, I keep the complete output. So very often it is possible to find errors of the register allocator or any other phase of the code processing within the compiler.

The above example of the failed tests is just one of many during the process of the compiler testing. I log the successful and unsuccessful tests in two independent files. The files are created for every directory that is tested. Each file with the results has a special header, which stores data necessary for the test archiving as can be seen in the following example:

```
arch:codasip_urisc
opt:2
ca/ia:-ca
sys:linux64
distro:CentOS 6.7x86_64
dir:int
version:6.2.0-0.j.1277.n.160728
result:failed
file_operations2.c simulation error
file_operations.c simulation error
920113-4.c compiler error
optargs-6.c assembler error
pr41981-1.c linker error
200897127-8.c compiler error
```

In the example above there is shown the list of failing tests. For each test it has an identification of the exact phase where the test failed. It is a simulation error in two cases, a compiler error in two cases and one assembler and one linker error. For each of the failing tests a log is kept and also an archive containing all the temporary files and outputs.

From the header it is clear that the testing was performed in the directory `int` and also what architecture was tested and the version of the testing tools. Moreover, the header contains information about the system of the testing and bit width.

6.2 Generator of the test selection files

The mechanism that is explained above has met the needs of our research project. However, as in our project we very often add new models and branches that need to be tested, we also need a way how to easily create a new file, that modifies the test usage, or to modify the files that already exist.

The best way for doing so, is to create a generator of such files. The generator would need the information about the tested platform as well as about the tests themselves. It would also very nicely fit into my plans about the high level of automation of the testing process. In the following subsection I will introduce such a generator.

6.2.1 Design of the generator of test selection files

The main task of the generator will be the creation of new `.x` files and also update of the existing ones. The generator will need the information about the platform that includes mainly:

- *bit width* - Is the platform 16/32-bit or does it have a different size?
- *availability of the libraries* - Do we have a compiler-rt library or any other library for the given model?

- *availability of instruction and cycle accurate description* - What level of description do I possess?

This is the main piece of information which I need to get about the platform. The majority of such information can be easily gathered. I will have a look at various possibilities in the implementation part of the generator.

The knowledge that I need to have from the side of the tests is a little bit less complicated. I just need to know what header files the test includes. I can say that if the test includes any header file, such as the test below, I need to generate a corresponding file. The test below will require the presence of the `Newlib`, as well as the presence of the `compiler-rt`.

```
#include <stdio.h>
#include <math.h>
#include <float.h>
#include <double.h>

double res(float i, double j){
    double res;
    res = M_PI*i*i*j;
    return res;

}

int main(){
    float i = 3.14159;
    double j = 4.9685;

    double res = mul(i,j);

    printf("%d", res);
    exit(0);
}
```

But the situation is not that straightforward. Certain tests might rely on availability of the library and not include any header files. Moreover, modern compilers in such situations do not exit with the error code, but just emit a warning and compile the test if the header file is available.

It seems that the only proper way how to find out if the test needs the support of any library for the given architecture is to compile the file and to find the necessary information from the temporary files.

The information I need can be obtained from several sources. I can get it either from the assembler format or from the object file. It is also possible to link the object files and in the case of an unsuccessful link, I get a list of missing symbols.

The last possibility is the least convenient one. It requires almost the whole process of compilation, that in the case of larger files can take several seconds. It also means that I have to compile the tests without the libraries to find out what symbols are needed and find the corresponding libraries.

Another possibility is to compile the test into the assembler format by the `llc` and to try to find the symbols in the assembler file. In the following lines, I can see the format of the assembler file.

```
$tmp4:
    CALL $__addvdi3
    LOAD R10, R0 + 40                // 4-byte Folded Reload
    LOAD R8, R0 + 48                 // 4-byte Folded Reload
    LOAD R9, R0 + 44                 // 4-byte Folded Reload
```

We can see that there is a call of *addvdi3* function. However, this function can be from another source file or it can be even a call of function in the same file. This information cannot be obtained at this phase of compilation. Moreover, the processing of the assembler files is quite difficult as there are no other tools that could provide the necessary information in our project.

The other possibility is to use the object file format. It is necessary to have the source file in the assembly language processed by the assembler and get the object file. This requires just one more compilation step in comparison to the previous case. However, the additional compilation step will give us a lot of useful information that was not available in the assembly language format.

The most desired information is if there are undefined symbols in the currently compiled module. This information can be obtained via tools, such as *objdump*. Below there is an example of the object dump output with given parameters.

```
addvdi3_test.o:     file format elf64-mips_basic

SYMBOL TABLE:
g_str   000000000000 info_string10_addvdi3_test.s
...
00000000000000d0 l       .text  000000000000 tmp15_addvdi3_test.s
00000000000001ac l       .text  000000000000 tmp27_addvdi3_test.s
0000000000000208 l       .text  000000000000 tmp33_addvdi3_test.s
0000000000000000 l       .debug_ranges  000000000000
   debug_ranges0_addvdi3_test.s
000000000000031c l       .text  000000000000 tmp53_addvdi3_test.s
000000000000037c l       .text  000000000000 tmp60_addvdi3_test.s
0000000000000000 l       .debug_info    000000000000
   cu_begin0_addvdi3_test.s
0000000000000000 l       .text  000000000000
   @debug_text_start_addvdi3_test.s_addvdi3_test.s
0000000000000000 l       .text  000000000000 @csl_.text_addvdi3_test
   .s
0000000000000000 l       .debug_frame   000000000000
   @debug_text_cie_pointer_addvdi3_test.s_0_addvdi3_test.s
0000000000000000 l       .text  000000000000
   @debug_text_fde_start_addvdi3_test.s_1_addvdi3_test.s
0000000000000040 l       .debug_frame   000000000000
   @debug_text_cie_pointer_addvdi3_test.s_2_addvdi3_test.s
00000000000000c8 l       .text  000000000000
   @debug_text_fde_start_addvdi3_test.s_3_addvdi3_test.s
0000000000000400 l       .text  000000000000
   @dwarf_retval_end_addvdi3_test.s_7_addvdi3_test.s
0000000000000000         *UND*  000000000000 __addvdi3
00000000000000c8 g     F .text  000000000338 main
0000000000000000         *UND*  000000000000 printf
0000000000000000 g     F .text  0000000000c8 test__addvdi3
```

We can see from the example that there are many defined symbols. To mention just some of them I can name, for example `main` or `_addvdi3_test.s`. Moreover, we can see that the file contains `.debug_ranges` and `.debug_info` sections that are used by the debug tools, such as debugger.

Nevertheless, most importantly I can easily identify the undefined symbols, which in this case are `__addvdi3` and `printf`. This indicates that I will have to link the compiler-rt library together with the standard C language library.

I have shortened the example as it was quite long and it would not fit the page. Some irrelevant symbols and information has been left out.

Once I have the needed information about symbols and what libraries should be linked, I need to generate a new file or update the existing one. This should not be a difficult task. For the implementation I have chosen the Python language.

I have called the tool for the generation of the `.x` file the *Constraintgen*. The implementation of the tool was performed in the Python language and the framework *pytest* [45].

One of the main advantages of the pytest is that it collects all the files with the prefix or suffix test and executes them. It also uses the system of fixtures [84], which is a system of dependencies. These dependencies create a hierarchy that is resolved by the pytest framework.

For the implementation, I had to create a set of fixtures. The fixtures are responsible for the generation of the file, creation of the toolchain that is able to compile the source file and the compilation of the source file to the object format.

Once a single test file is compiled, the object format generator fixture parses the object file and resolves dependencies. After the resolution is finished, the resulted constraint file is generated. There are also other fixtures, such as the reporter or the model, but these fixtures play a subsequent role.

For the scheme of the system see Fig. 6.2.

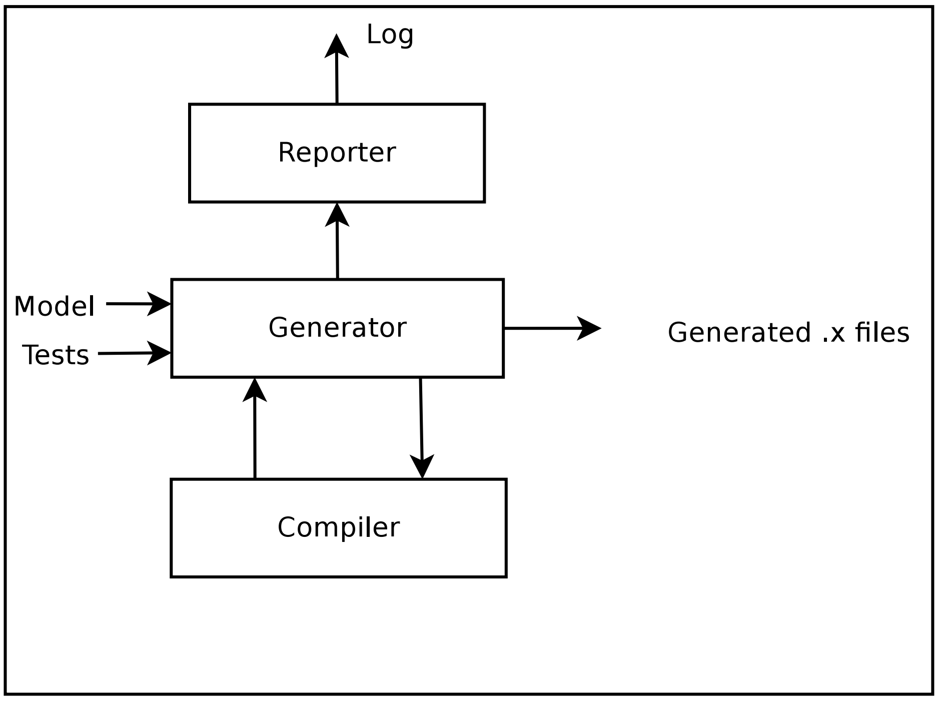

Figure 6.2. Scheme of the constraint generator

The inputs of the system are the directory with the model in the ADL language CodAL and the directory which contains the test, for which the `.x` files should be generated. This offers a possibility to create yet another layer above the *Constraintgen* that would offer an even higher level of automation.

6.3 Experimental results and contribution

With the implementation of the test selection generator *Constraintgen*, I have performed several tests. In Table 6.1, I have summarised a number of generated files for the MIPS and the Codasip uRISC core.

Core	*Number of tests*	*Number of generated files*
MIPS	1644	392
Codasip uRISC	1644	364

Table 6.1. Number of generated files

From the table, it is apparent that the number of tests is equal for both cores and the number of generated `.x` files is also comparable. The difference in the number of generated files is given by the fact that, in some cases, the compiler generates the call of the compiler-rt function while for the other core the call in not necessary. In both cases the majority of the files was generated because of the compiler-rt. The number of tests that required the Newlib library was lower.

The next Table 6.2 shows the speed of the generation.

Core	*Number of generated files*	*Number of folders with tests*	*Time of generation*
MIPS	392	9	84.11s
Codasip uRISC	364	9	77.64s

Table 6.2. Speed of the generation

From Table 6.2 we can see that the speed of the generation is very good. The speed of the generator is approximately 5 `.x` files per second, which I consider very good. Should the `.x` file be created by hand, it would take approximately 10 seconds for the creation of a single file.

The graph depicting the results of the *Constraintgen* is shown in Fig. 6.3.

The major contributions of the selected solution are as follows:

- *flexibility* - The tests from various test-suites are supported, there is no dependency on the test source, so this system can be used for simple tests as well as for benchmarks.
- *higher level of automation* - The files that are used during the test selection are generated fully automatically without a user interference.
- *scalability* - The system can be used for any new core, the generator is able to gather all the necessary information from the compiler automatically.
- *acceleration of the testing* - The tool is able to generate the files fast.

The system of the `.x` files, which can be used for the test selection was published in the journal article [30]. The article sketches the scheme of the files.

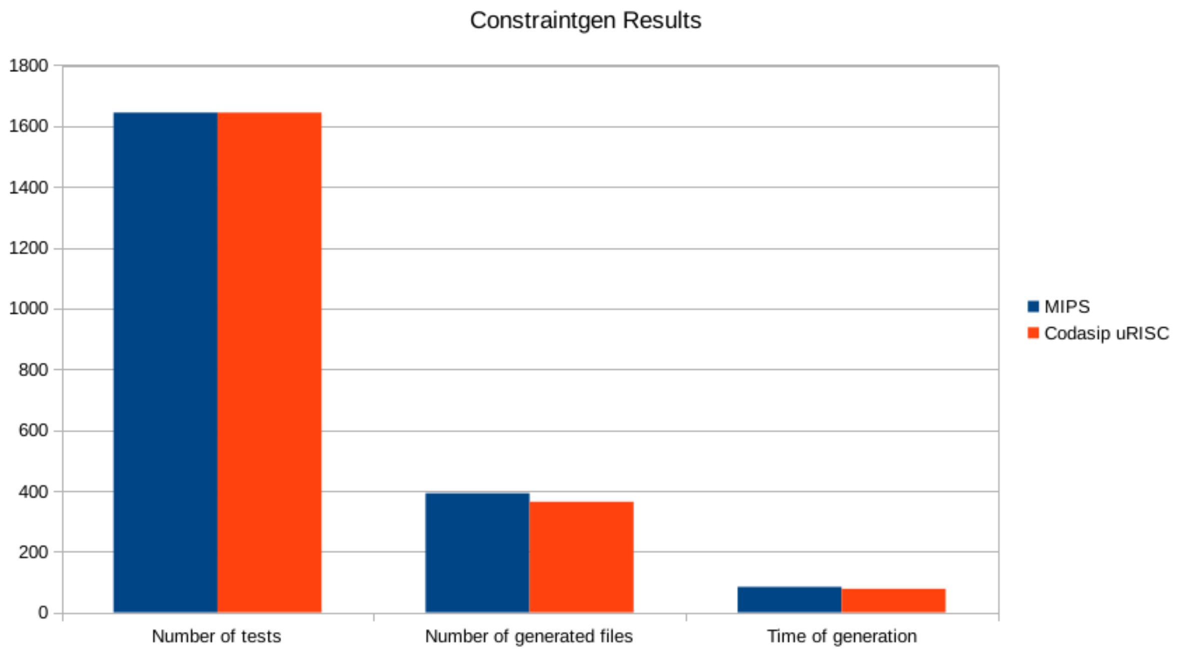

Figure 6.3. Results of the constraintgen

Chapter 7

Acceleration of testing

In this chapter, I will discuss the speed of the testing. As was mentioned at the beginning of the thesis, there is a big pressure for deployment of new builds more than once a day. I will focus mainly on the acceleration of the testing [28], [31] as the build acceleration was the focus of the thesis by Lukasova [67] that I supervised.

7.1 Testing attitudes

The testing of various parts of the project is very time consuming. I perform various types of tests that have different time demands. I have spent some time by reorganization of the tests and investigating whether I can utilize the results between the various tests.

7.1.1 Testing oriented on tools

In the tools oriented testing, we need to ensure that the generated tools as well as the generators themselves work properly. So both these parts need to be tested thoroughly. There are also interesting interconnections between the generators and the generated tools that can save a lot of computer time.

Let us have a look at the generators first. The generators are in our case triggered via a command line interface. I have created a set of classes that enable us to perform full tests of the command line functionality in the Python language. This test-suite, in combination with various models, gives us a very strong tool for ensuring that our generators are stable. The test-suite is highly modifiable. I can also very easily enhance this test-suite with performance tests and stress tests. The test-suite can be executed in a mode which tests all combinations of the parameters that are legal. However, this is very time consuming and I often test only certain combinations of parameters. The results of the generators testing is one of the inputs into the testing of the generated tools. The scheme is pictured in Fig. 7.1

When I get to testing of the specific generated tool, I first have a look at the tests of the generators. If I find out any problems during the generation, I either skip the tests as a whole or I need to pay more attention to the results of the testing.

After testing of the generators is finished, tests of the generated tools are executed. At this phase, it is possible to use the results of the generators testing. Because I have the results from various platforms, I can schedule and perform a test of the given tool, for example, the assembler only on the working platforms.

If there have been issues with generation on all platforms, I skip the whole process of testing. If the tool has been generated correctly, I put the generated binary under tests.

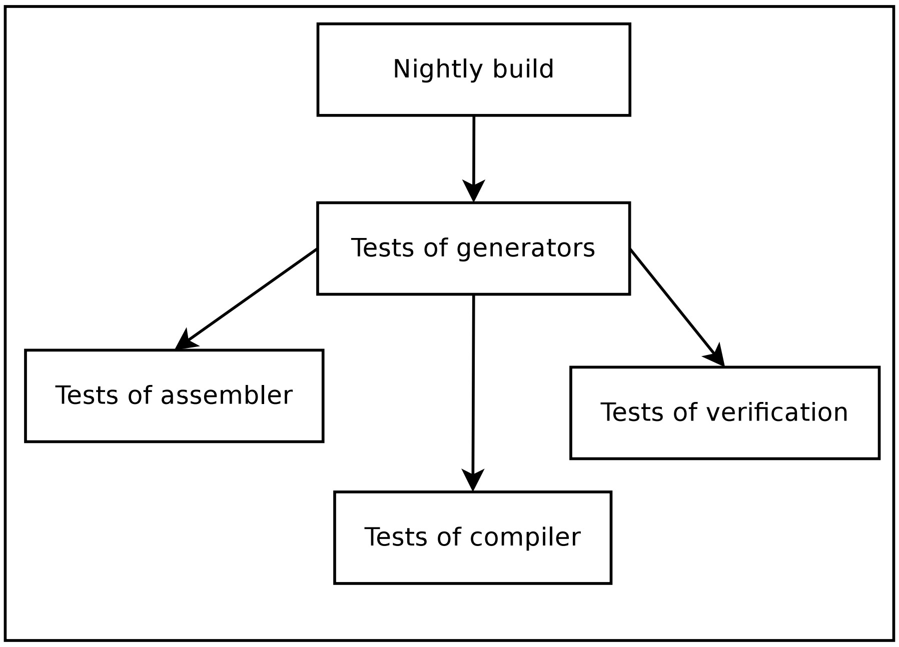

Figure 7.1. Scheme of the tools generation tests

In the case of the assembler, it is thoroughly tested by the `randomgen`. The `randomgen` program generates valid programs automatically from the processor description, which the assembler binary must be able to process. The `randomgen` application is also automatically generated so the paragraph above also applies to it. However, this is only one method of assembler testing. In this way I ensure that the valid constructions will be assembled without problems. Nevertheless, the assembler is also tested within the compiler driver.

Now I will have a look at testing of the compiler `backend`. The input of the `backend` are the files that are in a certain kind of internal representation of the compiler driver and the output is the assembly code. Here it is possible to see the very close interconnection with the assembler, which is responsible for transformation of the assembly language to the object files. I have several ways of testing the compiler `backend`. The first line consists of simple tests taken from various test-suites, such as the GCC torture test-suite. These simple tests are meant for fast debugging of the `backend`.

There I can also utilize the results of the generators testing. Not only that I have to check that the `backend` together with the compiler driver were generated, but I can also check if the necessary libraries, which are needed by the compiler, are available. If not, I can choose only the subset of tests and shorten the testing time. If I do not have the Newlib library compiled, I can save up to several hours of testing. The time savings are also achieved thanks to the test selection mechanism, which allows automatic detection of the libraries.

The second line consists of benchmarks. The purpose of these tests is to tune the performance of the compiler. They can also be used for the debugging, but it is not as comfortable as in the case of the simple programs mentioned above. What is important in this case is the fact that I very closely observe the number of cycles that are needed for each benchmark. If I have a rapid growth in the number of cycles, it indicates severe issues in the compiler and can lead to increased power consumption, which is unwanted in the cores for embedded systems.

The last set of compiler tests are really complex tests, such as the Linux core. This category serves as the ultimate test that the compiler, as well as the model, contains the minimum of errors. The results of the generators testing comes to use in this case as well. In addition to all the tools that are required for the tests of simple programs, I also require the presence of the Newlib library. For execution of all three categories of the programs a simulator is used.

I have introduced a scheme of the utilization of the tools generator results on the compiler. Nevertheless, I think that it will give us the biggest time savings in the case of verifications. The reason for this is the fact that there is a large number of verification tests and they are time consuming.

7.1.2 Testing oriented on models

Another point of view of the testing system is from the angle of the models. The model developer expects that the tools work without problems. They are interested in their processor design and need to get the results of testing all in one place. Therefore, their use case is completely different.

Errors in the model are very often revealed in the phase of tool generation. The tools contain various checks that ensure that the tool can be generated. For example, the compiler `backend` cannot be generated when the model does not contain certain instructions, for example `jump` and so on. Generally I can say that for the models oriented point of view, the generator testing is very important. The most model oriented tests, which I currently deploy, cover the area of functional verification.

The role of functional verification is to verify the equivalence of the instruction accurate (IA) and cycle accurate (CA) model, which were described above. There are also formal methods [18], but they are not currently used in our project. The IA model describes the controller on the level of instructions, while the CA model is more precise. It describes a set of operations that represents the separate actions between the clock cycles. From each description a tool is generated. In the case of IA, I generate the simulator, and in the case of CA, I use the generated verification environment. I execute the same program on both and then I compare the results. Such tests are performed when both model descriptions are stable as it uses tools from the IA and CA description. These tests help us to discover differences in model descriptions.

One of the drawbacks of this attitude is the time demand. The test environment, which is generated from the CA description, is very slow and the number of tests is vast. It is not uncommon for these tests to take more than 24 hours.

Nevertheless, here I can also utilize the knowledge I have from the testing fo generators. Moreover, I need the results from the compiler testing as I use the compiled binaries for execution.

7.2 Case study and experimental results

I will demonstrate the whole process on the tools generator and tests of generated tools for one of the cores. The whole process is triggered by the nightly build, as demonstrated in Fig. 7.1.

The job that is responsible for the nightly build is called simply `Build-Framework`. This job, once it is finished, triggers the job which is called `Toolchain-generator-codasip_-`

`urisc`. This job is responsible for performing tests of the generators. It performs all the necessary tests and produces a file with results in the form - test name: `result`. Should I have a set of tests with the names `fu-systemc`, `fu-verilog`, `fve-vhdl`, `fve-systemverilog`, the file with the results would have the following content:

```
fu-systemc:pass
fu-verilog:fail
fve-vhdl:pass
fve-systemverilog:pass
```

Once this job is finished, it triggers a build of other jobs based on the result file of the `Toolchain-generator-codasip_urisc`. The job, which is responsible for that, is called `Sorter`. The role of this job is to process the result file from the generator job and trigger the corresponding downstream jobs. This is pictured in Fig. 7.2.

The trigger of the job is connected to the checkout of repositories and the download of the saved artifacts from the previous jobs. The checkout and download of the artifacts can mean hundreds of megabytes. The jobs that are triggered as downstream jobs perform the functional verification. I trigger three jobs that perform the verification for the Verilog, VHDL and the SystemC language.

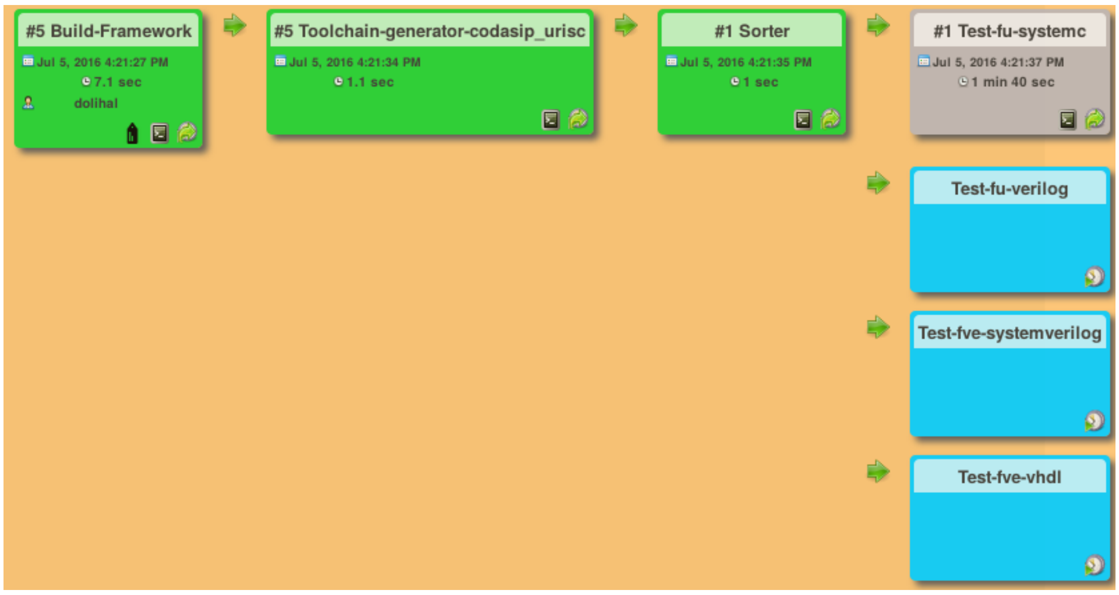

Figure 7.2. Build pipeline with tools generator

I will present the results of the testing which was performed within the Jenkins environment. The results were gained from the Jenkins server in version 1.652.

I have made several experiments with the utilization of the tool generator results and without it. I have also tried various combinations of the successful and unsuccessful jobs. I will present them in several tables and graphs below.

The results in the following Table 7.1 compare the time that was needed for tests of the functional verification with and without the use of the tools generation results.

The times in Table 7.1 do not include the time needed for the `Build-Framework` job. It is just the time needed for the testing. From the times we can see that the acceleration is apparent in all cases. The speed-up is gained by the fact that in the case of unsuccessful

Use toolsgen results	*Number of fails*	*Time*
YES	0	159m
NO	0	165m
YES	1	106m
NO	1	113m
YES	2	53m
NO	2	62m
YES	3	3m
NO	3	12m

Table 7.1. Comparison of the testing times.

generation of the environment, I do not have to download files from the git repository and also I do not have to copy large artefacts. The times, when the results of the generator tests were used, do include the time needed for performing the generator tests.

In the case of success, I also significantly reduce the size of the artefacts I have to copy, because I use pre-generated artefacts from the tests of generators. If I do not deploy the tools generator before the main tests, I have to generate a verification environment every time. A graph demonstrating the time savings is in Fig. 7.3.

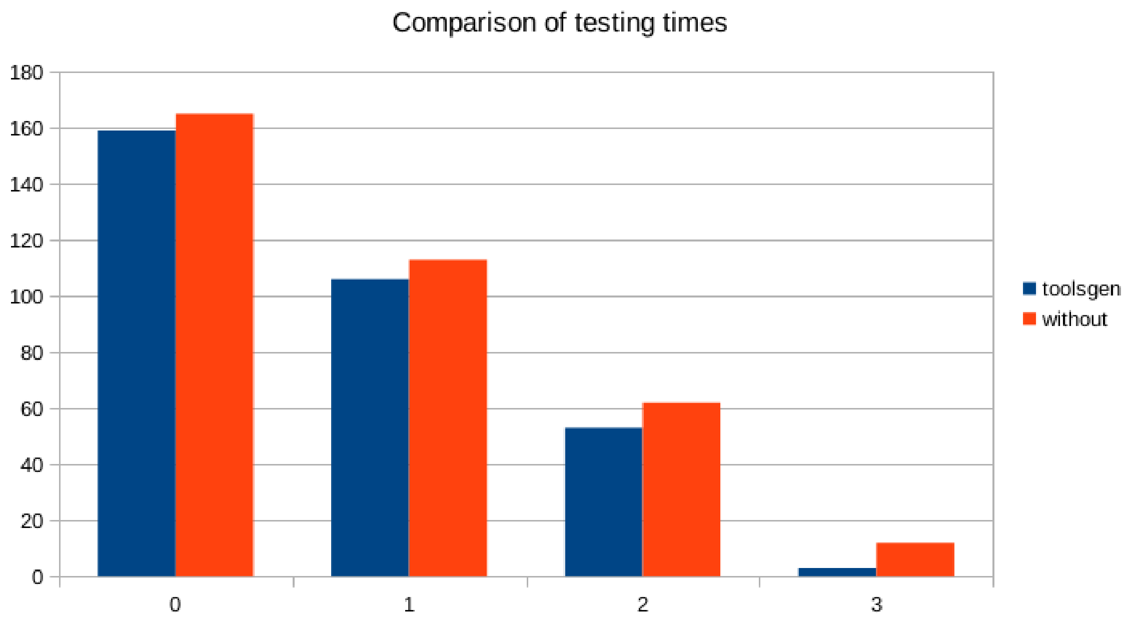

Figure 7.3. Comparison of the testing times

Table 7.2 shows the amount of data that has to be downloaded either from the git repository or as an artefact in the form of an archive. The sizes are important as they have a close relation to the time that is needed for testing. The more data has to be downloaded, the bigger overhead at the beginning of the testing.

From Table 7.2 it is clear that if jobs that do not use the results and the pre-generated environment, the amount of data is approximately 10 times bigger. What is even worse is

Use toolsgen results	*Number of fails*	*Downloaded data*
YES	0	141 MB
NO	0	1470 MB
YES	1	96 MB
NO	1	1470 MB
YES	2	44 MB
NO	2	1470 MB
YES	3	0 MB
NO	3	1470 MB

Table 7.2. Comparison of the data downloads.

the fact that the data has to be downloaded every time, even when the test is going to fail. On the other hand, when I prepend the generator tests and utilize the results, the amount of data decrease rapidly. The graphical representation is in Fig. 7.4.

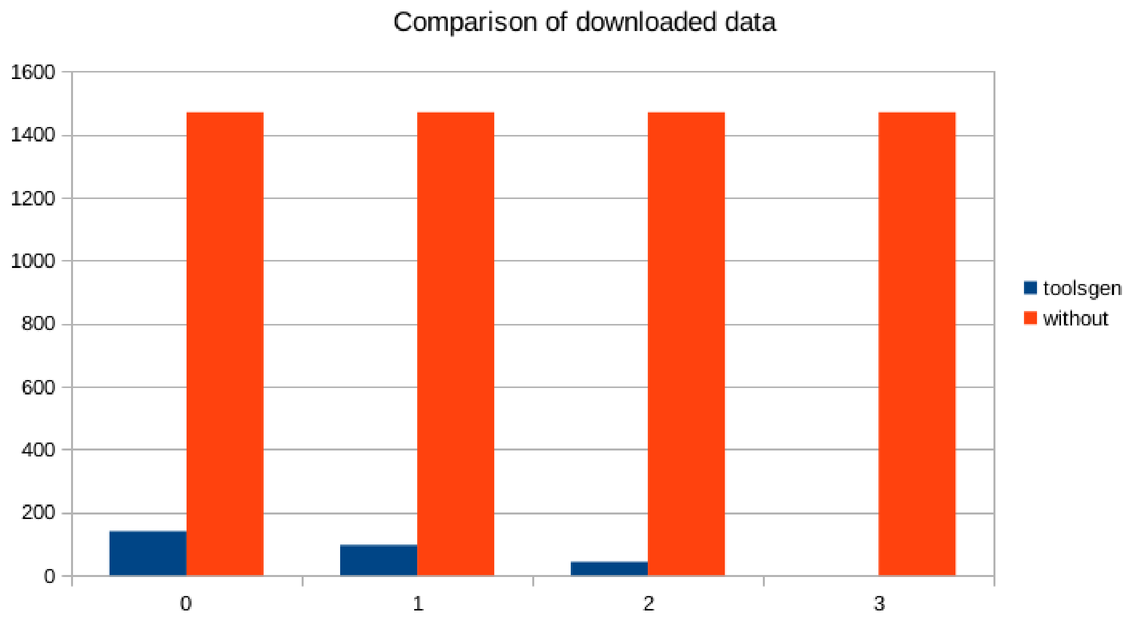

Figure 7.4. Comparison of the data downloads

7.3 Main contribution

The main contributions of the chosen approach are the following:

- *speed up of testing* - In the case of multiple failed jobs, the chosen approach can save a significant amount of time by not triggering the jobs that would fail, but even when the tests do not fail, the acceleration of tests is apparent.
- *traffic savings* - In the case of a failed job, the approach saves traffic as it prevents the checkout of repositories and the download of artefacts.

- *early notification* - The chosen approach gives the developer early notification regarding the state of the generators and generated tools, it presents in a clear form all features of the generator.
- *faster deployment* - In case I use the build automation described by [67], I will be able to deploy and test the new build more often than once a day.
- *possible reduction of the number of the triggered tests* - In the case of failed generator tests, the downstream jobs are not triggered.

The issues connected to the testing process were described in the article [31]. The use of the results of the generators tests was introduced in the article [28].

Chapter 8

Continuous integration job generator

In this chapter, I will address one of the greatest weaknesses of our project. I very often need to create a new set of tests for a new branch of a certain micro controller or create tests for a completely new core. In such situations, the user can create a whole new set of jobs by hand or find a way how to automatise such a task [29]. I have sketched the possibilities, which are provided by the plugins in the CI server Jenkins and also other solutions in the section State of art.

8.1 Jenkins continuous integration server

Continuous integration servers are a very popular solution for the automation of the tasks. The tasks usually fall into the categories of the build and the test automation. Nowadays there exists a wide range of solutions in the category of continuous integration servers. One of the most complete solutions is called the *Jenkins*.

The Jenkins is a continuous integration server that is supported by the community. It has a very swift pace of development and nowadays there exists a large number of plugins. Thanks to the plugins it is possible to add various functionality into the basic Jenkins server. Jenkins is not focused on just one single domain. With the correct choice of plugins, the user can build *Java, C*, *Python* and also other projects.

As far as the testing automation is concerned, the Jenkins environment provides support for execution of scripts for all the major scripting languages. Once the testing is finished, the server is also able to parse and visualise all major formats of the results.

The task that performs testing and is executed by the continuous integration server Jenkins is called the job. All the jobs are stored at the master server. Its configuration is stored in the form of the xml. Together with the xml, the server stores information about the latest builds. It keeps a history. The length of the history can be configured from within the Jenkins environment. The Jenkins server offers certain possibilities for automation of the job creation as was mentioned in the part called State of art. However, none of the possibilities suit my needs.

As the configuration is stored in a simple xml form, I thought of creation of the generator of the tests. Every time I needed to test a new core, I would run the generator of the tests and create a new set of testing jobs for the specific core. The generator should be lightweight. The specification of the job must be very simple and the configuration should be stored together with the model which we want to test.

Nevertheless, in order to create a generator of Jenkins jobs, I need to have good knowledge of the Jenkins job format.

8.2 Jenkins job format

Jenkins supports several types of jobs. The basic ones are the *freestyle project* and the *multiconfiguration project*. The main difference between the two is the fact that a multi-configuration project can be executed on multiple machines. There are also special types of jobs, which are tied to various plugins. There is the *maven job*, the *external job* or various views.

Below I have listed the basic description of the multi-configuration job, as it is the job which I am most interested in. Although I need to work with the other job types as well, the configuration of the job displayed below will be sufficient for the demonstration purposes now.

```
<?xml version='1.0' encoding='UTF-8'?>
<matrix-project plugin="matrix-project@1.4">
  <actions/>
  <description></description>
  <keepDependencies>false</keepDependencies>
  <properties>
    <com.sonyericsson.rebuild.RebuildSettings
   plugin="rebuild@1.22">
      <autoRebuild>false</autoRebuild>
    </com.sonyericsson.rebuild.RebuildSettings>
    <hudson.model.ParametersDefinitionProperty/>
  </properties>
  <scm class="hudson.scm.NullSCM"/>
  <canRoam>true</canRoam>
  <disabled>false</disabled>
  <blockBuildWhenDownstreamBuilding>false
  </blockBuildWhenDownstreamBuilding>
  <blockBuildWhenUpstreamBuilding>false
  </blockBuildWhenUpstreamBuilding>
  <triggers/>
  <concurrentBuild>false</concurrentBuild>
  <axes>
    <hudson.matrix.LabelAxis>
      <name>label</name>
      <values>
        <string>CentOS-6.5-32</string>
      </values>
    </hudson.matrix.LabelAxis>
  </axes>
  <builders>
    <hudson.tasks.Shell>
      <command>echo \$(pwd)</command>
    </hudson.tasks.Shell>
  </builders>
  <publishers/>
  <buildWrappers/>
  <executionStrategy class="hudson.matrix.
 DefaultMatrixExecutionStrategyImpl">
    <runSequentially>false</runSequentially>
```

```
40   </executionStrategy>
41 </matrix-project>
```

The whole configuration is in the xml format as was stated above. On the second line, we can see that it is the matrix project, which means that it can deploy multiple axes, and one of them is the configuration of the nodes. For simplicity, the job does not download any data from the Version Control Systems *VCS*. Another important tag is the one called `axes`, it is on line 22. This tells us that this job is built only on one node called `CentOS-6.5-32` on line 26. It is important to note that this job does not have parameters. If it had, the parameters would be visible at the top of the configuration.

There are also sections `builders`, line 30, and `publishers` on line 35. The section `builders` says that there is the shell script executed and the only command it runs is the *echo $(pwd)*. The job publishes no results, hence the part `publishers` is empty. The execution strategy is default. It is important to know what the configuration of the job looks like as I will work over the representation in the later sections.

8.3 Job generation

The main task that I need to deal with is the generation of the various jobs, which will ensure complex testing of the core. Mainly, I will generate the jobs which test the automatically generated tools. As I plan to control the whole system also from the command line, I wanted to avoid the graphical interface, at least in the first version of the project. I may add the graphical interface in the later versions, but I definitely need to keep the command line interface for the solution to be fully scriptable. This is also one of the reasons, why I cannot use the plugins provided by Jenkins. They have very poor documentation and are primary focused on usage via the web interface.

The basic scheme of my system is demonstrated in Fig. 8.1. We can see, that the whole system consists of just a few steps. The first part of the system is the *sniffer*. In my case it works over the *git repository*. Once the generation is triggered, the *job generator* uses *templates* to generate corresponding jobs. I will now give a more detailed description of the aforementioned parts.

8.3.1 Sniffer

I have decided to call this part of the generation process the *Sniffer* as it sniffs in the git repository for new branches. The main role of the Sniffer is to detect the creation of a new branch in the given git repository and trigger the generation. The whole system is designed in the way that the sniffer can be replaced by a different component. In the future, I would like to add support for other *VCS*. It also does not have to be present at all and can be completely omitted. The generator can be started by a different tool if it sticks to the defined interface.

Although currently the role of the Sniffer is to notify that a new branch has been created and deliver this information to the job generator. The Sniffer has no further intelligence and the whole system is designed in such a way that all decisions should be made in the generator itself. In the latest version, the Sniffer has a shape of the Unix script, which is executed repeatedly by the operation system.

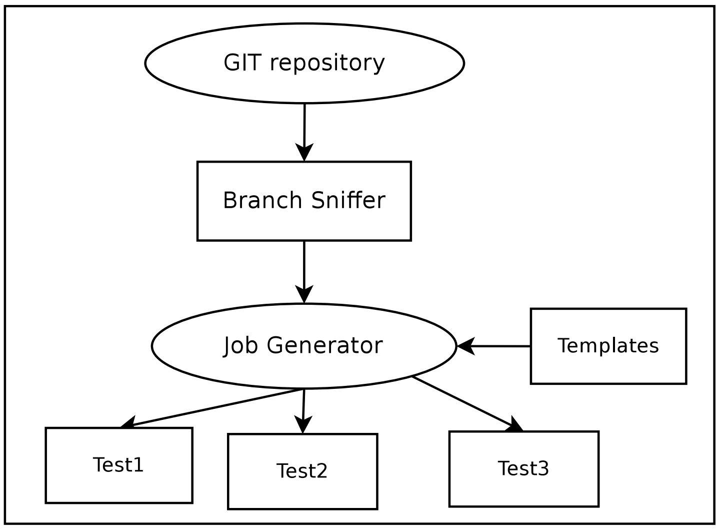

Figure 8.1. Scheme of the system

8.3.2 Templates

The second input into the job generator are the templates. I have various kinds of templates as I need to test various parts of the newly developed core. The main areas which have to be covered by test job generation are:

- compiler testing,
- functional verification,
- assembler testing,
- tools generation.

Please note that these are just the areas that need to be covered, not the jobs. Under each domain there is a variety of jobs which are generated and later on executed. There is usually just one template per domain, just in the case of functional verification I need to have several templates, as this area is very vast and I was not able to stick to just one template.

As far as the templates themselves are concerned, they are very simple. The templates are in the XML format, as are the jobs in the Jenkins, and the generated parts are in the form:

```
<string>@NODE_NAME@</string>
```

8.3.3 Job generator

Now when I have described the inputs of the generator, I will move to the generator itself. The job generator consists of several parts that are pictured in Fig. 8.2.

I decided to implement the generator in the Python language, because it allows very fast development, the code is very easy to read and the modifications are simple.

One of the first steps is the template selection. This part of the generator works over the configuration file that is present at the specific directory in the model branch which

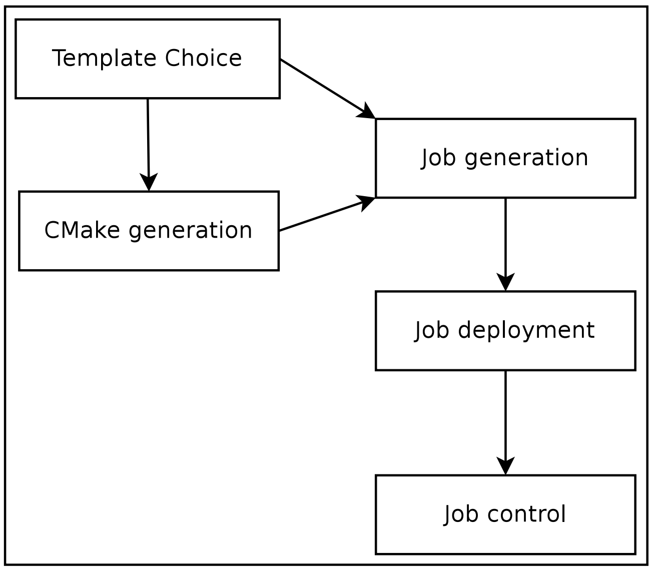

Figure 8.2. Scheme of the generator

should be tested. I have proposed a simple format of the configuration file that specifies the tested features. The other possibility I have is to automatically detect what features should be tested, but I have chosen the configuration file because some of the features cannot be automatically detected. From the specification file I am able to determine what templates should be used. The specification file has two major tasks:

- to define features that should be tested,
- to specify parameters for the generators.

However, the automatic detection of the features that should be tested was not completely abandoned. The detection is present but plays only a supplementary part.

Once the phase of the templates selection is finished, I need to generate the *CMake* files that will fill the desired information into the templates. *CMake* is a family of tools. These tools are designed for the build, testing and packaging of software. The generated *CMake* files are template specific as each template has different fields. Currently I generate one *CMake* file per template and I perform the generation in the separate directories.

From the two above mentioned inputs I can generate the job. The job generation is in fact just insertion of data into templates. I have decided to do this via the *CMake*, because it is one of the cleanest ways for doing so. The most frequent facts that are generated are the following:

- the branch used for testing,
- the node where the job is executed,
- the bash script and the parameters,
- the job name and the view where the job is placed.

The above mentioned information can be determined in the subsequently described way. The branch is one of the input parameters. It is delivered by the *Sniffer*, but it can also be delivered in a different way, it can be, for example, specified by the user.

The script, which is executed, could be a part of the template, however, this would increase the number of templates significantly. Therefore, I try to determine the name of the script. The name of the script can be determined from the information, which is given in the configuration file. Some of the scripts may have a variable number of parameters, but this I am able to determine from the directory structure of the model. Here I can see the supplementary part of the automatic detection.

The job name and view where the job should be placed are also determined from the configuration file and repository name. In the future I also plan to use a directory plugin in my installation, nevertheless, this should not be a problematic step.

The most complicated task is the selection of the correct node where the job should be executed. The management of the nodes is quite a complicated task and is described, for example, here [93]. There are certain jobs that can be executed only on specific sets of nodes. Typically, this is true for the jobs that perform tests of the functional verification or tests of the synthesis.

I have special groups of nodes, for example, for the execution of the verification jobs. The verification jobs require a preconfigured environment, which is present only on certain nodes, because the environment is very complicated. For such jobs, I have special templates with the predefined sets of nodes. Nevertheless, for the majority of jobs I do not have to solve such issues. I keep a simple table of nodes which is divided into sections which define what nodes are used for the specific jobs. I choose the jobs with the smallest number of assigned jobs and optionally I modify the assignment manually.

There is also other information that can be filled into the template. But the four above mentioned are the most common ones. I have the predefined default values for all the parameters that would suit most cases.

Very often I generate the parameters of the given job into the templates. They are stored in the *parameters* section and later these parameters are used in the *builders* section. However, there are also parameters that are node dependent. The node dependent parameters are defined in the Jenkins environment.

Frequently the generated job needs to use the artefacts from the other jobs. Nevertheless, I try to keep the generator as lightweight as possible and do not want to modify other jobs. The compatibility in this case is assured by the wild cards, and the name of the new job must fit into the wild card.

Once I have generated the jobs, which are needed for the testing of the newly developed branch, I have to upload these jobs to the CI server. For this purpose I use the Jenkins command line interface that performs the job upload and also registers the job.

When I create a job, which tests certain functionality, I also need to have a corresponding computer where the job will be executed. Some jobs require specially configured computers. For example, the jobs that test formal verification require the installation and configuration of special tools. However, the support for node management is very limited in the Jenkins environment. So I had to create a tool which helps me with this task.

8.4 Nodes management

One of the major problems I have faced in connection with the job generator was connected with the node selection and management. This problem became even more frustrating as I found out that the node management support is very limited in Jenkins. There is a plugin for *VirtualBox* but it is over three years old and does not support new versions. So there is no way how to administer nodes from within the Jenkins environment.

This is understandable when I take into account that Jenkins is the orchestrator and the nodes can be either physical machines or can be virtualized in any way. So it is left to the users of the environment to provide a solution that suits their needs.

In our project we use the virtualization software *VirtualBox*. From my point of view it is safer, and also more user friendly, to use virtualization methods. Moreover, I need to support quite a lot of operation systems. The goal of supporting multiple operation systems could not be achieved without the support of virtualization.

Therefore, I needed to design a solution that allows simple management of the virtual machines that I use for nightly builds and testing of the tools in our research project. I have cooperated on this part with Milan Skala, whose bachelor thesis [93] I supervised.

8.4.1 Design of the nodes management tool

Once the infrastructure for builds and tests has settled down and has been used for some time, I have identified the key tasks that the application for management should have. I try to summarise them in the following list:

- *support of the multiple platforms* - The program must run at least on the Unix and the Windows systems.
- *support of the multiple servers* - Where the virtual machines run.
- *grouping of machines* - The program must allow grouping of virtual machines and run of commands on such a group.
- *support of interactive mode* - Where the user can control the machine.

There were also some minor requirements, such as the support of configuration files and so on. Nevertheless, one of the very frequent tasks is the restart or upgrade of a certain group of machines. For example, we would like to update the kernel and restart all machines which have the operation system CentOS version 7.

I also need to take into account the support of multiple servers. I have several servers that I use just for virtualization of the build and the test machines, but I also have a large number of virtual machines that run on the user computers and during the night are used for testing purposes. That is the main reason why support of multiple servers is essential.

The main idea is to automatize the process of machines management as much as possible. Because of this, I do not need to have the graphical user interface in the first version of the tool. Management of the machines takes quite a lot of user time, so the pressure for automation is high. The tool must support the batch execution. The application must allow the definition of the tasks that should be performed on a given machine and leave the machine, once it is finished, without user interference.

The tool should be easily configurable. The configuration should support the configuration files with a given syntax and the parameters should also be passable via the command line interface. Thanks to this, the user will not be forced to perform the set up for every execution.

Once I have specified the requirements on the application, I can select the best method that will be used for implementation. The VirtualBox provides several possibilities. Although only one of the provided methods suits my needs. I must use the web service API as it is the only solution which supports virtual machines placed at different servers. On the other hand, it uses the XML format serialisation that is quite slow and has a negative impact on the performance of the whole solution.

The application contains three basic objects:

- *Group* - This class contains one or more virtual machines that can be placed at more physical servers.
- *Environment* - A class that contains information about the physical server. It keeps the information about the IP address and so on.
- *Interpreter* - The main class of the program, it executes the statements and keeps information about the environments and groups of computers.

One of the major requirements was batch execution without user interaction. In such a case the tool must be configured in such a way that it contains all the nodes which will be used during the batch file execution. The tool cannot ask the user for credentials that are needed for the connection to a certain node. The node can be stored in a configuration file where each line contains one node and the line has the following form:

```
/Deb-8-64 group=debian,64bit user=taylor password=t0ps3cret
```

The line means that the computer is at the server eva and has the name of Deb-8-64. It belongs to the groups debian and 64bit and has a given user and password. If the tool has the information from the configuration file, it does not have to ask the user.

The commands, which will be passed to the interpret, will be saved in a batch file that can be executed once all the nodes are added. The batch file will contain the statements that are understood by the interpret. The statements will be executed sequentially.

I can get into a situation when the batch file will contain unknown statements. For example, it will use virtual machines which are not configured or it will use the machines with an unknown user name or password.

I need to have a reliable way how to deal with such a situation. There are two basic scenarios I can either skip the operation and continue with the next statement or stop the execution immediately.

In this case, the best way is to inform the user about error via notification that is displayed on screen, create a log file with error detail and cease the execution. It is very probable that if any of the statements contained errors, it could affect the rest of the statements.

The diagram for execution of the batch file is in Fig. 8.3.

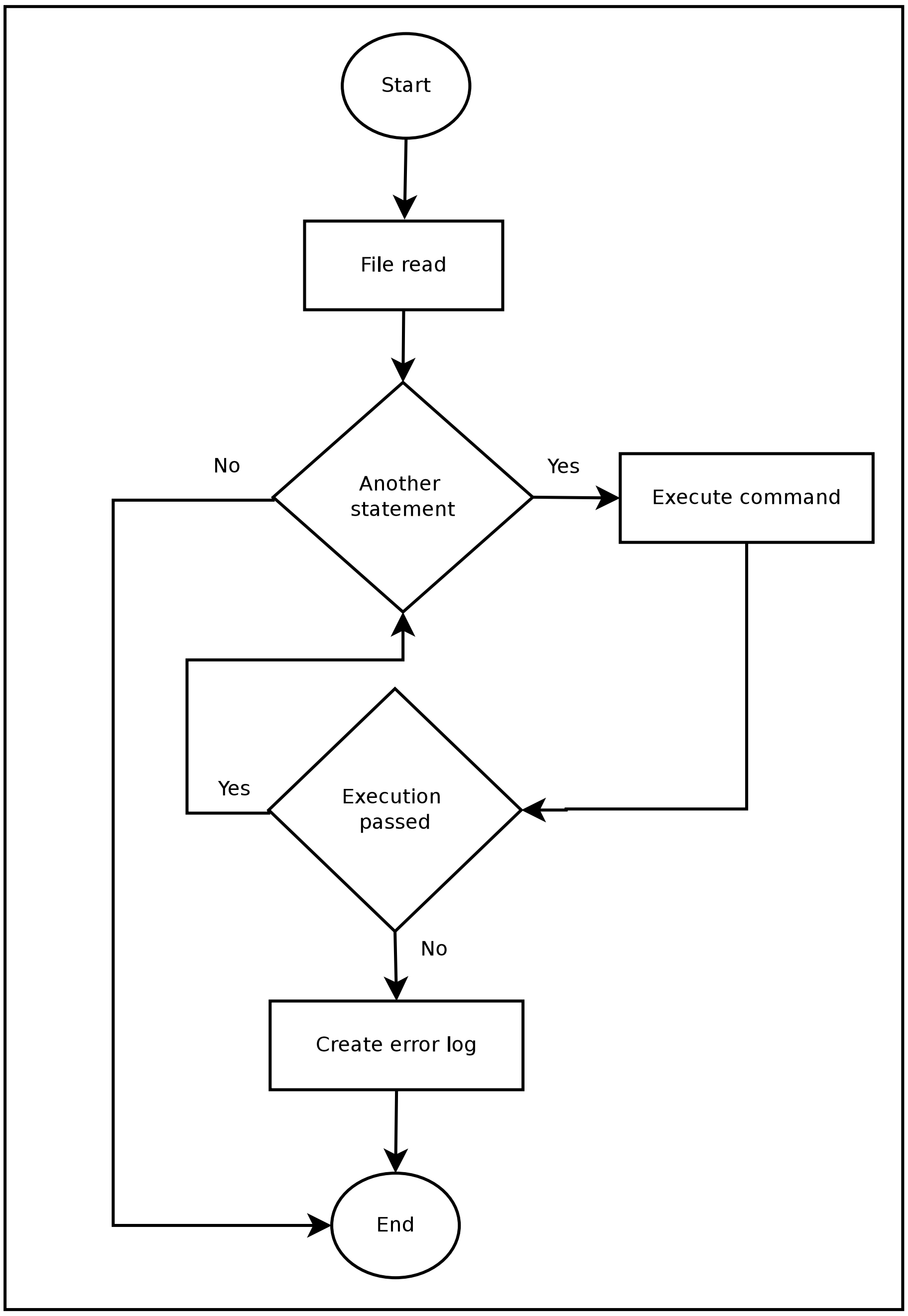

Figure 8.3. Scheme of the batch file execution

The implementation of the tools was performed in the Python language. Each class was implemented in a separate file. There are three main classes according to the basic objects. The class diagram of the solution is depicted in Fig. 8.4.

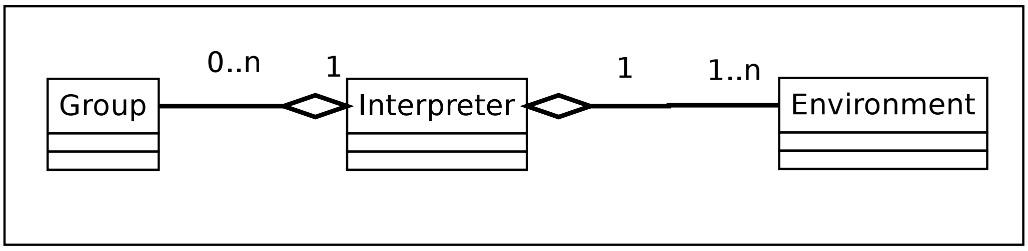

Figure 8.4. Class diagram for nodes management system

From the diagram it is clear that the instance of the interpreter does not have to contain a group of machines. On the other hand, it must contain at least one environment for the execution to begin.

With the tool for the management of the nodes, I can easily manage the machines which are used for the execution of the testing jobs. The tool is also used for the update of the file, which contains the nodes, which are used by the generator of the jobs.

8.5 Experimental results and contribution

With the current implementation of the simple job generator I have performed a number of tests. I have chosen two typical scenarios. The first case is the generation of a new testing set for the instruction accurate description of a new core. With the IA description corresponds the basic set consisting of tests which test the compiler and the assembler. When the complete description of the new core (instruction accurate, as well as cycle accurate) is created, the full set of tests is generated. The full set adds also tests for functional verification.

The templates, which are needed for the generation of such tests, were added into the template pool. The basic set consists of 3 jobs and the full set consists of 12 jobs. I have set the polling time to 6 minutes, so every 6 minutes the VCS server is polled for the new branches.

The times needed for the generation are summarised in Table 8.1. I have performed ten different runs, five for the basic set of tests and five for the full set of tests. The last row in the table shows the time which was needed for a run which was triggered manually.

Run	*Basic set*	*Full set*
1	84,1s	344,6s
2	208,5s	352,9s
3	154,9s	40,7s
4	110,5s	142,0s
5	51,3s	240,2s
Manual run	0,99s	4,2s

Table 8.1. Comparison of generation times.

In Table 8.1 we can see that the generation of three jobs takes 0.99 seconds, which gives exactly 0.33 second per job. When I try to generate the full set of 12 jobs, it takes 4.2 seconds. That is approximately 0.35 second per job.

All the jobs which I generate are multi-configuration jobs. The configuration of a multi-configuration job was described at the beginning of this chapter. The generation times vary for the basic set from 51 to 208 seconds. That is perfectly accurate, as the delay caused by the frontend is up to 360 seconds. The generation of the full set is also affected by the frontend delay and should be from 4.2 seconds up to 365 seconds. My measurements confirm that. I think that approximately 0.33 second is a very good generation speed. This time does not include the time needed for the upload of the new job to the Jenkins server via the command line interface. I have not included this time because it is largely affected by the position of the generating computer in the network and can also be heavily dependent on the network traffic.

I and my colleague have also tried to create the jobs manually. The group that created the jobs consisted of two persons. We tried to create the basic set of testing jobs and then the full set of jobs. The basic set of tests includes the generation of three jobs and covers the compiler and the assembler. The full set of jobs contains also jobs for verification. Together this set contains 12 jobs. Therefore, the sets are the same as in the previous measurement. I have also tried to compare the generation speed with the other generators provided by the Jenkins.

Method	*Basic set*	*Full set*
Lissom Generator	0,99s	4,2s
Jenkins job generator plugin	2,1s	8,5s
Jenkins DSL plugin	1,3s	5,2s
Manual creation	354s	1417s

Table 8.2. Comparison of creation times.

In Table 8.2 I have summarized the results of the generation and the manual creation. The manual creation of the jobs was the slowest in both cases.

The comparison with the most widely used generators provided by the Jenkins server was made at the following configuration. I used the Jenkins server in version 1.656. The Jenkins server was running on a server with 4 cores Intel i5 and has 8 GB of memory.

It is clear that the Lissom generator is faster than the job generator plugin and the DSL plugin in both tested cases. However, in the case of generation of just three jobs, the times are comparable. I have used the times from the manual run of my generator as both Jenkins plugins are also triggered manually.

In the case of generation of the big set, the Lissom generator has a clear advantage. It is 1s faster in comparison to the DSL plugin and 4.3 seconds faster in comparison to the job generator plugin.

The graphs depicting the generation times for the basic set and full set are in Fig. 8.5 and Fig. 8.6. The times for manual creation are not included.

The other advantage of the job generator is the fact that it is very lightweight and can be used for any kind of jobs. This largely depends on the templates that will be created. In theory we could completely abandon the creation of the jobs manually. If I provide the correct configuration of the jobs together with the set of templates, it is possible to generate the whole set of the testing jobs for any microcontroler.

Among the main contribution there can be placed:

- *significant speed up of the job generation* - As is clear from the results, the generation of the jobs is faster in comparison to any other generator.

- *higher level of automation* - With the correct configuration the job generation can be provided completely without user interference.
- *node management* - The tool provides functionality for nodes management, it is possible to create a new node and configure it for the given job.
- *wide range of use* - The job generator is dependent only on the xml format of the job, it can virtually generate any type of testing job.
- *no dependency on scripting language* - There is no need to deploy any scripting language, such as Groovy, the jobs are generated from the configuration file.

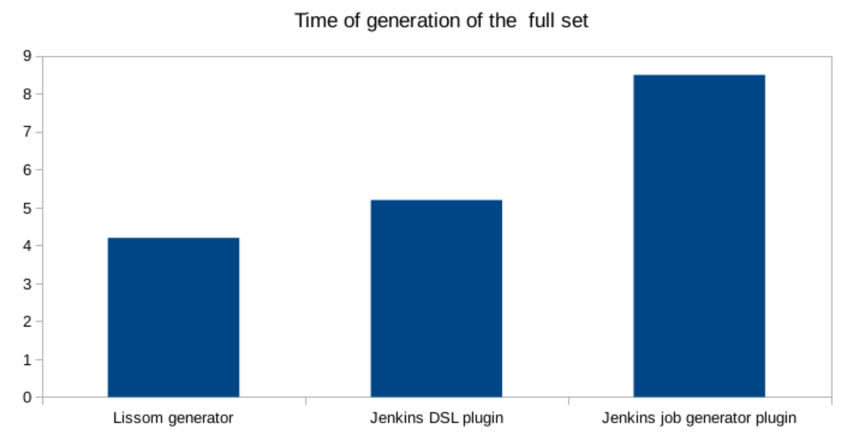

Figure 8.5. Graph of the full set generation

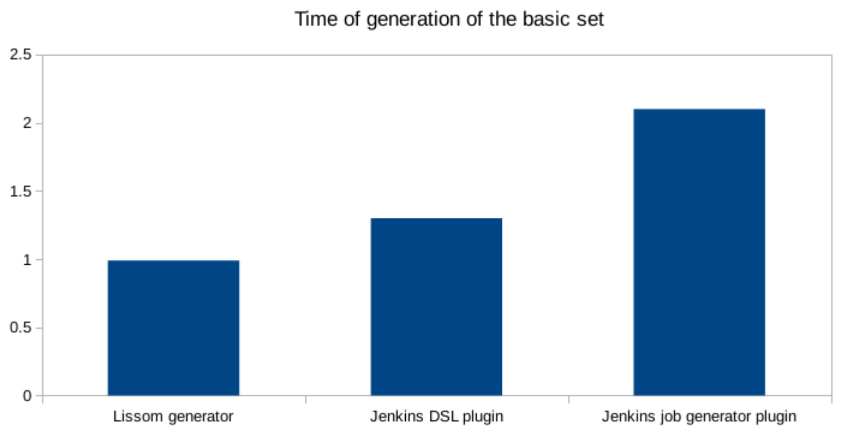

Figure 8.6. Graph of the basic set generation

The topic of the continuous integration environment and the automatic generation of the jobs for such environment was described in the article [29].

Chapter 9

Conclusion

In this thesis, I have addressed the testing of an automatically generated compiler. I have focused on four areas and introduced solutions that help to optimize and automatize the process of testing.

The first area is support of the standard C language library and the process of porting. Due to a good choice of the library, I was able to significantly increase the number of tests that can be used for porting. The raised number of tests gives the developer of the micro controller better possibilities for tuning the compiler and the whole system. I have introduced the universal mechanism that can be used for porting to any platform if the platform is suitable for the C library.

I have also worked on the process of porting with the aim to make it more automatic. I have introduced several ways that make the process of porting more automatic. The number of files that have to be manually changed has been significantly decreased and the whole process of porting is now faster and requires less knowledge.

The second area I have investigated is focused on the test selection mechanism. As was demonstrated, there is currently no mechanism that would suit my needs for the efficient selection of the test cases. I have designed a system of special files that are used for the selection of tests. The scheme is lightweight and robust at the same time. It can be used for any kind of tests and is not platform dependent, so it can be used for any core.

Moreover, I have created a generator of test selection files, which can be used for the generation of new files. The generator can be used once a new core, or just a new version of the existing core, is under development. The generator uses as an input the information contained in the model and the tests themselves that are compiled to the object form. The generation is fast and the accuracy of the results is good.

The area number three is connected with the acceleration of tests which are executed by the continuous integration server Jenkins. I have looked for a way how to decrease the time and space requirements of the functional verification testing and other tests. I have utilised the new kind of tests in our project, the tests of generators. The generator tests are executed as first, and all other tests use the results of the generator tests and, therefore, save time via the pre-generation of the binaries if the tests are successful. If the generator tests fail, the downstream jobs performing the verification tests are not triggered at all and hence save time and space that would otherwise be spent on the checkout of files.

Last but not least, I have sketched a simple generator of the Jenkins jobs that would suite our needs in the Lissom project. I need a generator that can be started by various ways, which is lightweight and can generate all kinds of jobs. This was one of the basic requirements, which was not met by any plugin that is currently available for the Jenkins.

I also wanted the tool to be at least partly independent of the Jenkins as it is not rare that the plugins do not cooperate well.

The current implementation of the generator is dependent just on the internal representation of the job. This is not a problem, as it is very simple to deploy new templates. At the same time, the internal job representation is not likely to change as it would imply changes in all plugins currently used by the Jenkins.

I put the generator under tests and the gathered results are very positive. As far as the speed of the generator is concerned, it cannot be matched by any tool that is currently available.

The implementation of the generators and other tools was performed in the Python language, so the solutions are easily extensible.

9.1 Future work

In the future, I would like to apply the use of the tool generator results also on other kinds of testing, such as the compiler or the assembler. I believe that I could gain some time savings in the case of application. Via this approach it should be possible to achieve speed for every group of tests that is more complex.

The implementation of generator of the testing jobs could also be extended. I could add support for the copy artefacts section and also support for the folders plugin that we currently use in our project. I would also like to find ways how to improve the speed of the generation.

It would also make sense to introduce a code generator into the testing process. It could uncover interesting new bugs in the automatically generated compiler.

Bibliography

[1] Job DSL Plugin.
`<https://wiki.jenkins-ci.org/display/JENKINS/Job+DSL+Plugin>` (July 2016), 2016.

[2] Job Generator Plugin.
`<https://wiki.jenkins-ci.org/display/JENKINS/Job+Generator+Plugin>` (July 2016), 2016.

[3] Template Project Plugin.
`<https://wiki.jenkins-ci.org/display/JENKINS/Template+Project+Plugin>` (July 2016), 2016.

[4] ACE: SuperTest Rembrandt Release - Update 2. User Documentation, 2014.

[5] ACE: CoSy compiler development system.
`<http://www.ace.nl/compiler/cosy.html>` (December 2015), 2015.

[6] ACE: SuperTest compiler test and validation suite.
`<http://www.ace.nl/compiler/supertest.html>` (Fedruary 2016), 2016.

[7] Aggrawal, K.; Singh, Y.; Kaur, A.: Code coverage based technique for prioritizing test cases for regression testing. *ACM SIGSOFT Software Engineering Notes*, year 29, nr. 5, 2004: pp. 1–4.

[8] Aho, A. V.; Ganapathi, M.; Tjiang, S. W. K.: Code Generation Using Tree Matching and Dynamic Programming. *ACM Trans. Program. Lang. Syst.*, year 11, nr. 4, October 1989: pp. 491–516, ISSN 0164-0925, doi:10.1145/69558.75700.
URL `<http://doi.acm.org/10.1145/69558.75700>`

[9] ANSI: INCITS/ISO/IEC 9899-1999 (R2005). `<http://webstore.ansi.org/RecordDetail.aspx?sku=INCITS/ISO/IEC%209899-1999%20%28R2005%29/>` (April 2016), 2016.

[10] ARM: ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition, Issue C. 2014.

[11] Arquilian: Arquilian. `<http://arquillian.org/>` (March 2016), 2016.

[12] Ashenden, P. J.: *The Designer's Guide to VHDL, Volume 3, Third Edition (Systems on Silicon) (Systems on Silicon)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., third volume, 2008, ISBN 0120887851, 9780120887859.

[13] Autotest: Autotest. `<http://autotest.github.io/>` (March 2016), 2016.

[14] Balarin, F.; Chiodo, M.; Giusto, P.; aj. (editors): *Hardware-software Co-design of Embedded Systems: The POLIS Approach*. Norwell, MA, USA: Kluwer Academic Publishers, 1997, ISBN 0-7923-9936-6.

[15] Bennett, J.: Howto: Porting newlib, A Simple Guide. 2010.

[16] Binder, R.: *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[17] Chandra, R.; Dagum, L.; Kohr, D.; aj.: *Parallel Programming in OpenMP*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, ISBN 1-55860-671-8, 9781558606715.

[18] Charvat, L.; Smrcka, A.; Vojnar, T.: Automatic Formal Correspondence Checking of ISA and RTL Microprocessor Description. In *Proceedings of the 13th International Workshop on Microprocessor Test and Verification (MTV 2012)*, Institute of Electrical and Electronics Engineers, 2012, ISBN 978-1-4673-4441-8, pp. 6–12. URL `<http://www.fit.vutbr.cz/research/view_pub.php?id=10135>`

[19] Codasip: CodAL Manual. Technical report, Codasip, Brno, CZ, 2014.

[20] Csmith: Csmith bug database. `<https://embed.cs.utah.edu/csmith/gcc-bugs.html>` (August 2016), 2016.

[21] Cucumber: Cucumber. `<https://cucumber.io//>` (March 2016), 2016.

[22] Cytron, R.; Ferrante, J.; Rosen, B. K.; aj.: Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, year 13, 1991: pp. 451–490.

[23] Davis, F. D.; Bagozzi, R. P.; Warshaw, P. R.: User acceptance of computer technology: a comparison of two theoretical models. *Management science*, year 35, nr. 8, 1989: pp. 982–1003.

[24] De Micheli, G.; Rolf, W., E.and Wolf: *Readings in Hardware/Software Co-design*. Morgan Kaufmann, 2001, ISBN: 9781558607026.

[25] Dolihal, L.; et al.: Use of Architecture Description Language ISAC fo ASIP Design. In *In Proceedings of Eighth International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems*, European Network on High Performance and Embedded Architecture and Compilation, 2012, ISBN 978-90-382-1987-5.

[26] Dolihal, L.; Hruska, T.: Porting of C library, Testing of generated compiler. In *In Proceedings of The Sixth International Multi-Conference on Computing in the Global Information Technology*, International Academy, Research, and Industry Association, 2011, ISBN 978-1-61208-008-6, pp. 125–130.

[27] Dolihal, L.; Hruska, T.: Semiautomatic Porting of the C Library. In *In Proceedings of International Conference on Computer Science, Computer Engineering, and Education Technologies*, International Academy, Research, and Industry Association, 2014, ISBN 978-1-941968-02-4, pp. 86–89.

[28] Dolihal, L.; Hruska, T.: Overview of the testing environment for the embedded systems. In *In Proceedings of The third International Conference on Green Computing, Technology and Innovation*, International Academy, Research, and Industry Association, 2015, ISBN 978-1-941968-15-4, pp. 86–89.

[29] Dolihal, L.; Hruska, T.: Automatic Job Generation for Compiler Testing, Testing of Generated Compiler. In *In Proceedings of The Eighth International Conference on Advances in System Testing and Validation Lifecycle*, International Academy, Research, and Industry Association, 2016, ISBN 978-1-61208-500-5, pp. 1–6.

[30] Dolihal, L.; Hruska, T.; Masarik, K.: Testing of an automatically generated compiler, Review of retargetable testing system. In *International Journal on Advances in Software, 2012*, year 2012, International Academy, Research, and Industry Association, 2012, ISSN 1942-2628, pp. 15–26.

[31] Dolihal, L.; Hruska, T.; Masarik, K.: Testing System for the HW/SW Codesign Toolchain. In *In Proceedings of Eighth Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, NOVPRESS, 2012, ISBN 978-80-87342-15-2.

[32] Dolihal, L.; Hruska, T.; Masarik, K.: Usage of simulators in testing system. In *In Proceedings of Industrial Simulation Conference 2012*, EUROSIS, 2012, ISBN 978-90-77381-71-7.

[33] Donald, T.; Moorby, P.: *The Verilog Hardware Description Language*. Springer, 2002, ISBN 978-1402070891.

[34] Duggal, G.; Suri, B.: Understanding regression testing techniques. In *Proceedings of 2nd National Conference on Challenges and Opportunities in Information Technology*, Citeseer, 2008.

[35] Duvall, P.; Matyas, S. M.; Glover, A.: *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional, 2007, ISBN 0321336380.

[36] Eclipse: Eclipse. `<https://eclipse.org/>` (August 2016), 2016.

[37] Fauth, A.; Van Praet, J.; Freericks, M.: Describing instruction set processors using nML. In *In Proceedings of European conference on Design and Test*, IEEE Computer Society Washington, 1995, pp. 587–593.

[38] Fowler, M.; Foemmel, M.: Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006: p. 122.

[39] Gatliff, B.: Porting and Using Newlib in Embedded Systems. `<http://neptune.billgatliff.com/newlib.html>` (March 2016), 2016.

[40] GCC: GCC Compiler website. `<https://gcc.gnu.org/>` (Fedruary 2016), 2016.

[41] Git: git. `<https://git-scm.com/>` (February 2016), 2016.

[42] Halambi, A.; Grun, P.; Ganesh, V.; aj.: EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability. In *In Proceedings of the Design, Automation, and Test in Europe*, Springer Netherlands, 2008, pp. 485 – 490.

[43] Hat, R.: Red Hat. <https://www.redhat.com/> (August 2016), 2016.

[44] Hoffmann, A.; Meyr, H.; Leupers, R.: *Architecture exploration for embedded processors with LISA*. Kluwer, 2002, ISBN 978-1-4020-7338-0, I-VIII, 1-230 pp.

[45] Hubertz, J.: *Softwaretests mit Python*. Springer, 2016, ISBN 978-3662486023.

[46] Huizinga, D.; Kolawa, A.: *Automated Defect Prevention: Best Practices in Software Management*. ISBN 0470042125, 9780470042120.

[47] Husar, A.: *Programming of reconfigurable systems using a higher programming language*. Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2014.

[48] ISO/IEC: Working Draft, Standard for Programming Language C++, N3337. 2011.

[49] Jenkins: CloudBees Folders Plugin. <https://wiki.jenkins-ci.org/display/JENKINS/CloudBees+Folders+Plugin> (June 2016), 2016.

[50] Jenkins: Git Plugin. <https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin> (June 2016), 2016.

[51] Jenkins: Jenkins. <https://wiki.jenkins-ci.org/display/JENKINS/Home> (March 2016), 2016.

[52] Jenkins: Matrix Authorization Strategy Plugin. <https://wiki.jenkins-ci.org/display/JENKINS/Matrix+Authorization+Strategy+Plugin> (June 2016), 2016.

[53] Jenkins: Role Strategy Plugin. <https://wiki.jenkins-ci.org/display/JENKINS/Role+Strategy+Plugin> (June 2016), 2016.

[54] Kroustek, J.: *Retargetable analysis of machine code*. Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2014.

[55] Krzikalla, O.: Performing Source-to-Source Transformations with Clang. <llvm.org/devmtg/2013-04/krzikalla-slides.pdf> (August 2014), 2013.

[56] Labs, S.: Sauce Labs. <https://saucelabs.com/> (March 2016), 2016.

[57] Leroy, X.: Formal Verification of a Realistic Compiler. *Commun. ACM*, year 52, nr. 7, July 2009: pp. 107–115, ISSN 0001-0782, doi:10.1145/1538788.1538814. URL <http://doi.acm.org/10.1145/1538788.1538814>

[58] Leung, H. K.; White, L.: Insights into regression testing [software testing]. In *Software Maintenance, 1989., Proceedings., Conference on*, IEEE, 1989, pp. 60–69.

[59] Leupers, R.: *Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, ISBN 0792379896.

[60] Leupers, R.; Marwedel, P.: *Retargetable compiler technology for embedded systems: tools and applications.* Norwell, MA, USA: Kluwer Academic Publishers, 2001, ISBN 0-7923-7578-5.

[61] Lier, F.; Wienke, J.; Wrede, S.: Jenkins for FloBI–A Use Case: Jenkins & Robotics. In *Jenkins User Conference*, 2013.

[62] Lindig, C.: Random testing of C calling conventions. In *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, ACM, 2005, pp. 3–12.

[63] Lissom: Project Lissom Webpages. `<http://www.fit.vutbr.cz/research/groups/lissom/>` (August 2014), 2014.

[64] LLVM: LLVM Language Reference Manual. `<http://llvm.org/docs/LangRef.html>` (December 2015), 2015.

[65] LLVM: LLVM Compiler website. `<http://llvm.org/>` (Fedruary 2016), 2016.

[66] LLVM: The LLVM Compiler Driver (llvmc). `<http://llvm.org/releases/2.2/docs/CompilerDriver.html>` (March 2016), 2016.

[67] Lukasova, M.: *Build Paralelization in Jenkins Environment.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 201.

[68] Mackinnon, T.; Freeman, S.; Craig, P.: Endo-testing: unit testing with mock objects. *Extreme programming examined*, 2001: pp. 287–301.

[69] Marwedel, P.: The Mimola design system: Tools for the design of digital processors. In *Proceedings of the 21st Design Automation Conference*, IEEE Press, 1984, pp. 587–593.

[70] Masarik, K.: *System for hardware-software codesign.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2008.

[71] McKeeman, W. M.: Differential testing for software. *Digital Technical Journal*, year 10, nr. 1, 1998: pp. 100–107.

[72] MiBench: MiBench. `<https://github.com/embecosm/mibench>` (June 2016), 2016.

[73] Microsoft: Microsoft. `<https://www.microsoft.com/>` (August 2016), 2016.

[74] Mishra, P.; Dutt, N. (editors): *Processor Description Languages.* Morgan Kaufmann, 2008, ISBN 0-12-374287-0.

[75] Mondal, S.: *Compiler Back End Generation from nML Machine Description.* Master's Thesis, Indian Institute of Technology, Kanpur, 1999.

[76] Muller, P.: *Automoatizovane metody hledani chyb v prekladacich.* Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2008.

[77] Newlib: Newlib. `<https://sourceware.org/newlib/>` (March 2016), 2016.

[78] Oquendo, F.: π-ADL: an Architecture Description Language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. *ACM SIGSOFT Software Engineering Notes*, year 29, nr. 3, 2004: pp. 1–14.

[79] Palka, M. H.; Claessen, K.; Russo, A.; aj.: Testing an Optimising Compiler by Generating Random Lambda Terms. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0592-1, pp. 91–97, doi:10.1145/1982595.1982615. URL `<http://doi.acm.org/10.1145/1982595.1982615>`

[80] Perennial: Perennial C Compiler Valication Suite. `<http://www.peren.com/pages/products_set.htm>` (August 2014), 2015.

[81] Perennial: Perennial test suite. `<http://peren.com/>` (Fedruary 2016), 2016.

[82] Prikryl, Z.: *Advancem Methods of Microprocessor Simulation*. Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2011.

[83] Prikryl, Z.; Kroustek, J.; Hruska, T.; aj.: Fast Just-In-Time Translated Simulation for ASIP Design. In *14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2011, ISBN 978-1-4244-9753-9, pp. 279–282. URL `<http://www.fit.vutbr.cz/research/view_pub.php?id=9567>`

[84] pytest: pytest fixtures: explicit, modular, scalable. `<http://pytest.org/latest/fixture.html>` (July 2016), 2016.

[85] QEMU: QEMU: Open Source Processor Emulator. `<http://www.qemu.org/>` (March 2016), 2014.

[86] RISC-V: User-Level ISA Specification v2.1. 2016.

[87] Rowen, Chris and Hennessy, John , and Christensen, Clayton M. and Leibson, Steve: *Engineering the complex SOC : fast, flexible design with configurable processors*. Prentice Hall Modern Semiconductor Design Series, Upper Saddle River: Prentice Hall, 2004, ISBN 0-13-145537-0. URL `<http://opac.inria.fr/record=b1108184>`

[88] Selenium: Selenium. `<http://www.seleniumhq.org/>` (March 2016), 2016.

[89] Shaw, K.: Generating New Jenkins Jobs From Templates and Parameterised Builds. `<http://www.blackpepper.co.uk/generating-new-jenkins-jobs-from-templates-and-parameterised-builds/>` (July 2016), 2012.

[90] Simkova, M.: *New methods for increasing efficiency and speed of functional verification*. Master's Thesis, Faculty of Information Technology, Brno university of Technology, 2015.

[91] Simkova, M.; Prikryl, Z.; Hruska, T.; aj.: Automated Functional Verification of Application Specific Instruction-set Processors. *IFIP Advances in Information and Communication Technology*, year 4, nr. 403, 2013: pp. 128–138, ISSN 1868-4238. URL `<http://www.fit.vutbr.cz/research/view_pub.php?id=10268>`

[92] Siska, C.: A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *In Proceedings of the 11th International Symposium on System Synthesis*, 1998, pp. 31–36.

[93] Skala, M.: *Virtual Machine Management System*. Master's Thesis, Faculty of Information Technology, Brno university of Technology, 201.

[94] Smith, M. D.: Machine SUIF Project website. `<http://www.eecs.harvard.edu/hube/software/software.html>` (December 2015), 2015.

[95] Sreedhar, V. C.; Ju, R. D.-C.; Gillies, D. M.; aj.: Translating Out of Static Single Assignment Form. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, London, UK, UK: Springer-Verlag, 1999, ISBN 3-540-66459-9, pp. 194–210.
URL `<http://dl.acm.org/citation.cfm?id=647168.718132>`

[96] Synopsys: IP Designer, IP Programmer and MP Designer. `<http://www.synopsys.com/IP/ProcessorIP/asip/ip-mp-designer/Pages/default.aspx>` (August 2014), 2014.

[97] Synopsys: Processor Designer. `<http://www.synopsys.com/systems/blockdesign/processordev/pages/default.aspx>` (August 2014), 2014.

[98] Technologies, M.: MIPS32 Architecture For Programmers, Volume II: The MIPS32 Instruction Set. 2003.

[99] Teich, J.: Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 2012.

[100] VASG: VHDL Analysis and Standardization Group. `<http://www.eda.org/vhdl-200x/>` (December 2015), 2015.

[101] Views, D.: Software Development Costs: Bugfixing. `<http://blog.pdark.de/2012/07/21/software-development-costs-bugfixing/>` (March 2016), 2016.

[102] Wang, L.-T.; Chang, Y.-W.; Cheng, K.-T. T.: *Electronic design automation: synthesis, verification, and test*. Morgan Kaufmann, 2009.

[103] Wiki, G.: A Brief History of GCC. [Online] `<https://gcc.gnu.org/wiki/History>`, 2008.

[104] Xie, T.; Taneja, K.; Kale, S.; aj.: Towards a Framework for Differential Unit Testing of Object-Oriented Programs. In *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, IEEE Computer Society, 2007, ISBN 0-7695-2971-2.

[105] Yang, X.; Chen, Y.; Eide, E.; aj.: Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, New York, NY, USA: ACM, 2011, ISBN 978-1-4503-0663-8, pp. 283–294, doi:10.1145/1993498.1993532.
URL `<http://doi.acm.org/10.1145/1993498.1993532>`