CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF INFORMATION TECHNOLOGY

ASSIGNMENT OF MASTER'S THESIS

Title:	Design and implementation of a prototype virtual machine for a Component-based programming and modeling language Compo
Student:	Bc. Tomáš Licek
Supervisor:	Ing. Petr Špaček, Ph.D.
Study Programme:	Informatics
Study Branch:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2016/17

Instructions

Component-based languages represent a potential future for development of large-scale information systems. Reflective languages then allow these systems to be easily manageable, thanks to introspection and intercession qualities that reflection offers. The aim of this thesis is to design and implement a prototype virtual machine for a reflective Component-based programming and modeling language called Compo.
1. Get familiar with the model and meta-model of Compo and also with prototype implementation in the Pharo Smalltalk environment.
2. Together with the author of a parallel Master's thesis (entitled similarly as this work, author Julius Soltes) design and implement a front-end transforming Compo expressions into an abstract syntax tree.
3. Design a virtual machine to run basic language constructs.
4. Using C or C++ language, implement a prototype of the virtual machine.
5. Evaluate the language constructs coverage provided by your prototype.

References

Will be provided by the supervisor.

L.S.

doc. Ing. Jan Janoušek, Ph.D.	prof. Ing. Pavel Tvrdík, CSc.
Head of Department	Dean

Prague December 10, 2015

Czech Technical University in Prague

Faculty of Information Technology

Department of Theoretical Computer Science

Master's thesis

Design and implementation of a prototype virtual machine for a Component-based programming and modeling language Compo

Bc. Tomáš Licek

Supervisor: Ing. Petr Špaček, Ph.D.

8th May 2016

Acknowledgements

I would like to thank my supervisor, Ing. Petr Špaček, Ph.D., for valuable advices and friendly and enthusiastic attitude all the time. I would also like to appreciate support of my parents and their toleration of my rapidly changing moods.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 8th May 2016 . . . .

Citation of this thesis

Licek, Tomáš. *Design and implementation of a prototype virtual machine for a Component-based programming and modeling language Compo.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstrakt

Tato práce se zabývá návrhem a implementací virtuálního stroje pro komponentově orientovaný programovací jazyk Compo. Komponentově orientované programování je obecně založeno na skládání jednoduchých komponent do komplexních systémů. Standardní programovací jazyky v dnešní době pracují v drtivé většině s objektově orientovaným modelem. Objektově orientované programování ovšem nenabízí potřebnou úroveň abstrakce. V případě převedení komponentového modelu na objektový model existuje riziko postupného vytracení požadované architekury systému. Komponentově orientované jazyky oproti objektovým umožňují explictní vyjádření architektury přímo v kódu. Komponentově orientované modely jsou v neposladní řadě o mnoho bližší lidskému myšlení.

Klíčová slova komponentově orientované programování, Compo, modelování systémů, virtuální stroj

Abstract

This thesis deals with design and implementation of a virtual machine for component-based programming language Compo. Component-based modeling is based on composition of trivial parts into complex systems. Ordinary programming languages are mainly object-oriented these days. Object-oriented programming does not offer needed level of abstraction. There is a risk of loss of an architecture model while trying to transform a component model to an object model. Component-oriented programming, in comparison to object-oriented programming, provides explicit expression of architecture directly in a code. Moreover, component-oriented programming is much closer to the human way of thinking.

Keywords component-based programming, Compo, system modeling, virtual machine

Contents

Introduction **1**
Related work . . . 1
Structure of the thesis . . . 2

1 Analysis **3**
1.1 Description of CBSE . . . 3
1.2 Current state of CBSE . . . 4
1.3 SCL, the predecessor of Compo . . . 9
1.4 Contribution of Compo . . . 10
1.5 Basic description of Compo . . . 11
1.6 Contribution of this thesis . . . 13
1.7 Summary . . . 14

2 Design **15**
2.1 Overview . . . 15
2.2 Used tools . . . 15
2.3 Used libraries . . . 17
2.4 Grammar extraction . . . 17
2.5 Frontend . . . 18
2.6 Bootstrapping . . . 20
2.7 Interpreter execution . . . 25
2.8 Memory . . . 26

3 Implementation **27**
3.1 Implementation notes . . . 27
3.2 Grammar implementation . . . 27
3.3 Abstract syntax tree structure . . . 33
3.4 Memory objects implementation . . . 36
3.5 Exceptions . . . 39
3.6 Bootstrapping implementation and structures . . . 40

3.7 Value components 42
3.8 System component 43
3.9 Inheritance 44
3.10 Interpreter implementation 45
3.11 Summary 48

4 Tests 49
4.1 Unit tests 49
4.2 Real-world examples 51
4.3 Valgrind report 51
4.4 Documentation 52
4.5 Tests summary 52

5 Future work 53

Conclusion 55

Bibliography 57

A Grammar 59

B New descriptors 65

C Acronyms 73

D Contents of enclosed CD 75

List of Figures

1.1 ArchJava compiler 7
1.2 Compo compiler 14

2.1 Architecture rule 18
2.2 AST for while 21
2.3 AST for descriptor 21
2.4 Component-Descriptor relationship 22
2.5 Clash of parent components 25

3.1 AST namespaces 33
3.2 AST inheritance 35
3.3 Visitor structure 35
3.4 Complicated component 37
3.5 Memory bootstrap 41

4.1 Code coverage 51
4.2 Valgrind 52

Introduction

Software development evolution is faster and faster every year. Evolution primarily lies in reusing solutions of sub-problems that were solved by someone else in the past. From the ordinary programmer point of view, the whole running computer with its complex circuits inside and its operating system is a sub-problem solved before. Deep in the history, ones and zeros were the only way to program computers. As programmers observed repetitive patterns in ones and zeros, they realized that these patterns could have some names and that they can be **reused**. These names represent the assembly languages. Same process of reusing patterns occurred with assembly languages and later with procedural languages.

Majority of today's production programming langauges is object-oriented. Object-oriented programming provides a decent way how to design and implement various types of software. However, there are some issues with common object-oriented languages. The biggest problem is that object-oriented languages rely on programmer's thoroughness and do not enforce correct structure of the code by itself. Therefore, new approaches of software engineering are studied and the natural step is to make a **reusage** more explicit directly in the code.

Current research shows that the component model is already known and present, but only in the design phase of software engineering. TIOBE survey [1] shows that the most popular languages are object-oriented languages like C++ or Java (and even Assembly language). Nevertheless, component programming languages are already available.

Related work

This thesis is based on dissertation thesis of Ing. Petr Špaček, Ph.D. [2]. The dissertation thesis thoroughly analyses Component-based Software Engineering (CBSE), goes through related existing technologies and mainly comes up with principles of Compo — custom component-based programming language. The dissertation thesis contains detailed description of Compo language. Many references of virtual machine functionality will be pointed into the dissertation thesis. A part of the dissertation thesis is a prototype implementation of a virtual machine for Compo written in Pharo — Smalltalk development environment. This thesis benefits from Smalltalk implementation mainly in extracting the actual Compo grammar. Other inspirations by Smalltalk implementation are less than humble.

Structure of the thesis

The text starts with Chapter 1, which contains brief exploration of current CBSE, presentation of selected existing languages and frameworks and in the end Compo language principles.

Chapter 2 contains description of structure of an existing Smalltalk prototype and overall description of the virtual machine architecture. This chapter also justifies selection of used technologies.

Chapter 3 goes deep into the implementation of the virtual machine and also explains various particular implementation decisions.

Chapter 4 is composed of description of test approaches and test reports.

Chapter 5 points out subjects for successors of this thesis.

Chapter 1

Analysis

This chapter covers the current state of CBSE research, presents existing technologies and talks over the principles of Compo language.

1.1 Description of CBSE

Decomposition is the most natural way to solve problems. It is present in almost every branch of industrial production. Clear example of decomposition is automotive industry. Every car has an engine, wheels, a steering wheel — all of these are components that are assembled together. The same approach is applied in CBSE.

CBSE is an approach that uses COP[1] to develop reusable components (*development for reuse*) and to assemble software from these reusable *off-the-shelf* components, connected together into various kinds of architectures (*development by reuse*).

Software components enable practical reuse of software *parts.* There are other units of reuse, such as libraries, designs, etc. As noted in [3]: "*software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system*". Benefits of reusing *off-the-shelf* components are obvious. Reusing improves quality and supports rapid development. At the same time, change of requirements leads to change of only affected components rather than the whole complex system.

Let us mention major advantages of CBSE:

- Reusability
- Maintainability

[1]Component-oriented programming is a programming technique and paradigm producing reusable components as the output of coding process.

- Clarity
- Flexibility

CBSE distinctly simplifes:

- Analysis of the system
- Understanging of how the system works
- Evolution of the system

1.2 Current state of CBSE

As mentioned in Introduction, CBSE is already present and known. Component-based approach opens the door for the large-scale application development and analysis. Current component-based approaches differ in many aspects.

One of the problems CBSE suffers from is that the component-based approach is not enforced by the most used — object-oriented — programming languages. Component-based approach means to follow some rules or to utilize available frameworks while using an object-oriented language such as Java or C++.

1.2.1 CBSE approaches

Following sections go briefly through three major categories of current CBSE approaches as described in [2].

1.2.1.1 Generative strategy

Generative strategy utilizes ADLs[2]. ADLs provide high-level overview of software system organization. An ADL describes the components which the system is composed of, interconnection among them and constrains how components interact. ADLs help during the design stage and aid in the communication of system designers. ADLs are usually able to generate basic code skeletons from architecture description. Architecture description is an abstract overview of the system, therefore ADLs are implementation-language independent.

However, there are some issues with ADLs. A big gap appears between the architecture description and the final shape of the system. As mentioned herein, ADLs are implementation-language independent. It means that it is not guaranteed that the design constraints submitted in the design stage keep in the implementation stage. Moreover, design constraints may completely vanish during system evolution and functionalities addition.

[2]Architecture Description Languages

1.2.1.2 Framework strategy

Framework strategy provides component models and development frameworks[3]. Majority of frameworks is provided for ordinary object-oriented languages. The software developer is able to achive quite good results using frameworks. Frameworks offer a sort of programming guidelines to build complex system from off-the-shelf components. The problem is that object-oriented language does not enforce correct structure of the code by itself. Despite the fact that development framework offers guidelines and a proper way for application development, a user of that framework is able to violate offered rules.

1.2.1.3 Component-oriented language strategy

Generative strategy and framework strategy use ADLs[4] or DSLs[5] to model or implement complex systems. There is a gap between these two approaches. Lack of built-in architecture constraints check results in difficulties during implementation, testing and software evolution stage.

COL[6] strategy is an evolution compatible with previous strategies. It is difficult to preserve the original idea while transforming design to implementation. COLs enforce both strategies directly in the code. COLs operate in a design domain and offer a natural way to build component-based systems. Such programming languages have support for component definition and composition.

Listing 1.1 demonstrates implicit involvement of moving strategy.

```
class MovingObject {
    private * AbstractStrategy;
    public MovingObject() { }
    public void setStrategy(AbstractStrategy * s) {...}
    public AbstractStrategy getStrategy() {...}

}
```

Listing 1.1: Moving object example.

The intended semantics is: "*class MovingObject requires an instance of custom implementation of AbstractMoveStrategy*". This example shows that the requirement for a moving-strategy is made implicit. It means that a user of this code may not be aware of this information without any knowledge of

[3]Software structure to support common programming tasks. Usually contains helper programs, API libraries, etc.

[4]Architecture Description Language

[5]Domain Specific Language

[6]Component Oriented Language

the internal structure of class or without any documentation. COLs made the requirement explicit by allowing developers to express full description of executable components.

The research in [2] showed that with COLs it is possible to bridge the gap between the design and implementation stage. Conformity of architectural constraints is automatically guaranteed, since these properties are captured directly in the code. The research also showed that COL approach brings better results in decomposition and decoupling.

Moreover, COL approach supports maintenance and evolution of software. For instance, generative approach is forced to regenerate an implementation code of an architecture design every time the design is changed. Managing software evolution and productivity are the main interests of MDE[7].

1.2.2 Selected existing technologies

This section shortly summarizes and describes some of the existing technologies concerning component approach.

1.2.3 ArchJava

ArchJava is a small, backwards-compatible extension to Java that integrates software architecture specifications smoothly into the Java implementation code. ArchJava seamlessly unifies the architectural structure and implementation in one language allowing flexible implementation techniques, ensuring treaceability between the architecture and the code, and supporting the co-evolution of the architecture and implementation. According to [4], ArchJava is intended to investigate the benefits and drawbacks of a relatively unexplored part of the ADL design space. ArchJava approach extends a practical implementation language to incorporate architectural features and enforces communication integrity. Key benefits are better program understanding, reliable architectural reasoning about the code and keeping the architecture and the code consistent as it evolves.

To allow programmers to describe software architecture, ArchJava adds new language constructs to support *compontents*, *connections* and *ports*. Example listed in listing 1.2 code is taken from [4].

[7]Model Driven Engineering

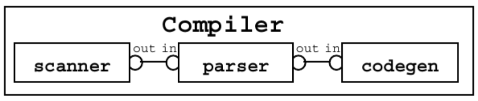


Figure 1.1: Graphical compiler structure

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol, SymTabEntry e);
    requires Token nextToken() throws} ScanException;
  }
  public port out {
    provides SymTabEntry setInfo(Token t);
    requires void compile(AST ast);
  }
}
```

Listing 1.2: ArchJava parser example.

The Parser component class uses two ports to communicate with other components in a compiler. The parser's *in* port declares a required method that requests a token from the lexical analyzer, and a provided method that initializes tokens in the symbol table. The *out* port requires a method that compiles an AST to an object code, and provides a method that looks up tokens in the symbol table.

```
public component class Compiler {
    private final Scanner scanner = ...;
    private final Parser parser = ...;
    private final CodeGen codegen = ...;

    connect scanner.out, parser.in;
    connect parser.out, codegen.in;

    public static void main(String args[]) {
        new Compiler().compile(args);
    }

    public void compile(String args[]) {
        // for each file in args do:
        ...parser.parse(file);...
    }
}
```

Listing 1.3: ArchJava compiler example.

The *Compiler* component class in figure 1.1 contains three subcomponents: a Scanner, a Parser, and a CodeGen. The scanner, parser, and codegen are connected in a linear sequence with the out port of one component connected to the in port of the next component.

1.2.3.1 CORBA

As described in [5], CORBA[8] is a technical standard for ORB[9]. ORB is a object-oriented version of an older technology called RPC[10]. ORB (or RPC) is a mechanism for invoking operations on an object (or calling a procedure) in a different (*remote*) process that may be running on the same, or a different, computer. At a programming level, these *remote* calls look like *local* calls.

CORBA is sometimes referred to as *middleware* or *integration software*. This is because CORBA is often used to get existing, stand-alone applications communicating with each other. CORBA is also an object-oriented distributed middleware.

CORBA utilizes definitions *client* and *server*, but CORBA is not as strict about these terms as other computer technologies. In CORBA terminology, a *server* is a process that contains *objects*, and a *client* is a process that makes calls to objects. A CORBA application can be both a client and a server at the same time.

An architectural aspect of CORBA is covered in its IDL[11]. An IDL file defines the public API[12] that is exposed by objects in a server application. The type of CORBA object is called an *interface*, which is similar in concept to a C++ class or Java interface. IDL interfaces support multiple inheritance.

Example in listing 1.4 is taken from [5].

[8]Common Object Request Broker Architecture
[9]Object Request Broker
[10]Remote Procedure Call
[11]Interface Description Language
[12]Application Programmig Interface

```
module Finance {
    typedef sequence<string> StringSeq;
    struct AccountDetails {
        string name;
        StringSeq address;
        long account_number;
        double current_balance;
    };
    exception insufficientFunds { };
    interface Account {
        void deposit(in double amount);
        void withdraw(in double amount)
            raises(insufficientFunds);
        readonly attribute AccountDetails details;
    }
}
```

Listing 1.4: CORBA example.

IDL interface may contain operations and attributes. Attributes are syntactic sugar for a pair of get- and set-style operations. Attribute can be read-only, in which case it maps just to a get-style operation. The parameters of an operation have a specified direction, which can be *in* (meaning that the parameter is passed from the client to the server), *out* (the parameter is passed from the server back to the client) or *inout* (the parameter is passed in both directions). Operations can also have a return value. An operation can *raise* (throw) an exception if something goes wrong. One can notice, that IDL is close to C++ language.

For further details on CORBA refer to [5].

1.3 SCL, the predecessor of Compo

The dissertation thesis [2] is based on previous work presented in [6]. SCL language is intended to be a minimal basis of a really usable component-oriented language. SCL was built to be minimal, simple, detailed and dedicated to CBSE. The core of SCL is built upon the following concepts: a component, a port, a service, a connector, a glue code, and the following mechanisms: port binding and service invocation. All of these properties are composed into programming language. A skilled programmer can develop independent, reusable components with it. A less experienced programmer can reuse previously created components and interconnect them into a new unit.

SCL utilizes class/instance approach and clearly distinguishes it. A component is a run-time entity and is an instance of a component *descriptor*.

A communication protocol is built on unidirectional named ports. Ports allow programmers to group a set of services and require or provide this set via required respectively provided ports. Components communicate by service invocation through their ports.

Areas of SCL improvement are explicit architecture description and reflection. SCL is hard to use as an architecture description language, because it focuses more on the functional aspect of components rather than on the specification aspect.

1.4 Contribution of Compo

As stated in [2], contributions of Compo language are as follows:

- **Uniformity**. Compo proposes an *everything is a component* operational development paradigm. The system is self-described by the explicit definition of the root of the instantiation tree (Descriptor) and the root of the inheritance tree (Component).

- **Architecture within implementation**. Compo tries to integrate smoothly a rich architectural description with a programming language to enforce full structural conformance between design and implementation. Compo provides architecture description constructs, so that developers can specify an architecture during designing and then fill in the architecture with the Compo implementation code.

- **Unique communication protocol**. In SCL, sending service invocation through a port is the only possible way for two components to interact. An effort was made to integrate an ownership relation to achieve hierarchical design, thus solution based on internal required ports is proposed.

- **Openness and extensibility**. Reflection provides the necessary levels of openness making the language uniformly accessible by the user. It opens the essential possibility that architectures, implementations and transformations can all be written at the component level and using a unique language. It encourages introspection and indeed adaptation of the underlying structure and behavior of the platform.

- **Modeling friendly inheritance system**. Reuse scheme designed for Compo is quite innovative in the context of CBSE, because it promotes modeling power with covariant specializations. Using *extends* statement, a new descriptor can be defined on the base of an existing descriptor, such a descriptor is then called a *sub-descriptor*. Sub-descriptors may introduce new ports or extend interfaces of inherited ports.

1.5 Basic description of Compo

Only a very basic description of Compo language could be found in this section. This thesis is not intended to be a complete reference of Compo language. For further details refer to [2].

As described in [2], Compo is inteded to bridge the gap between an object-oriented and an architecture-oriented design and also gather properties of existing, though insufficient, approaches into one complete and comprehensible language. Existing approaches were thoroughly examined in [2] and the result is that all approaches together abound with decent amount of new concepts, but these concepts are scattered around. For instance, both ArchJava (see [4]) and ComponentJ (see [7]) present mechanisms that allow the building of components, however, in ArchJava it is not possible, for instance, to export the behavior of interal components and to define new component structures at runtime. Unlike ComponentJ, which components are used to instantiate objects, ArchJava's components hold state variables, implemented methods and communication ports.

1.5.1 The language philosophy

Compo's philosphy is to keep the language as simple, minimal and uniform as possible, while at the same time incorporate all core concepts and mechanisms necessary for description and implementation of independent components and for description and implementation of high-level component-based architectures.

1.5.2 Concepts

This section benefits mainly from [2] as it is a complete reference for Compo language.

- **Component**: a run-time entity which provides and requires services through ports.
- **Descriptor**: an entity which describes the structure and the behavior of a particular kind of components in terms of declaration and definition of the external contract and the internal architecture.
- **Port**: a named communication and connection point; described by a name and a list of service signatures.
- **Service**: a unit of behavior definition.
- **Connection**: describes a binding from one to another port.

1.5.3 Mechanisms

- **Component creation (instantiation)**: a mechanism for building new components according to the description a descriptor defines. Such components are then called instances of the descriptor.

- **Service invocation**: a mechanism for run-time communication in between components.

- **Composition mechanism**: a mechanism for creating a new component by connecting off-the-shelf components within the context of the new component.

- **Substitution mechanism**: a mechanism for replacing components.

1.5.4 Example

Listing 1.5 shows a basic usage of Compo language. The same example as for ArchJava was selected for better understanding.

```
descriptor Compiler {
    provides {
      in  : { compile() }
      out : { getCode() }
    }
    internally requires {
      scanner : Scanner;
      parser : Parser;
      codegen : CodeGen;
    }
    architecture {
      connect scanner to default@(Scanner.new())
      connect parser to default@(Parser.new())
      connect codegen to default@(CodeGen.new())
      delegate in@self to in@scanner;
      connect out@scanner to in@parser;
      connect out@parser to in@codegen;
      delegate out@codegen to out@self;
    }
    service compile() {
      ...
    }
    service getCode() {
      ...
    }
}
```

Listing 1.5: Compo compiler example.

Descriptor *Compiler* externally (default visibility) provides two ports: *in* and *out*. Ports provide services *compile()* respectively *getCode()*. Descriptor internally requires three ports: *scanner*, *parser* and *codegen*. These ports are interconnected as stated in *architecture*. Interconnection is quite obvious but for clarity there is a diagram of interconnection shown in figure 1.2.

1.6 Contribution of this thesis

Contribution of this thesis is to explore options in designing and implementation of a pure component-based virtual machine. Until now, there is not such a real-world virtual machine. The closest concept is ArchJava (described in section 1.2.3), but it benefits mostly from Java language (inheritance and reflection). As described in [2], other component-based language approaches are similar to ArchJava: it is a component-oriented language build upon an object-oriented virtual machine.

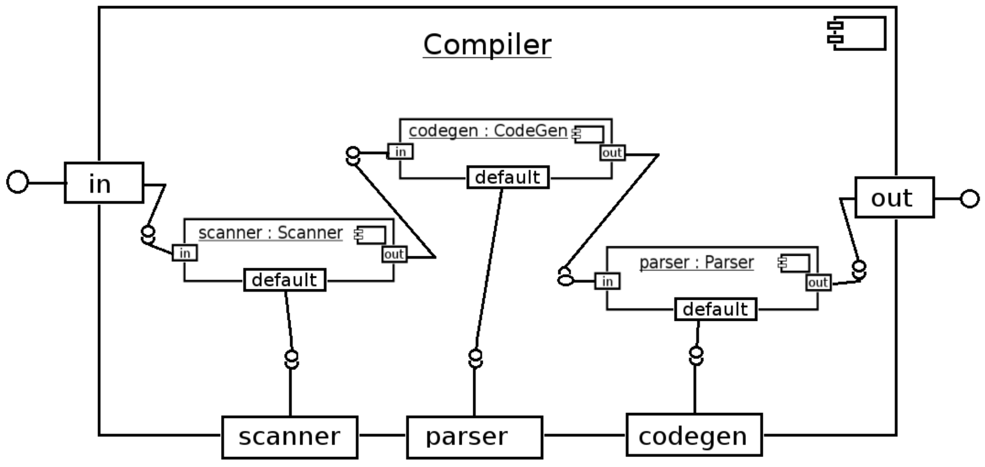


Figure 1.2: Graphical structure of compiler.

1.7 Summary

Motivation for using CBSE[13] as well as basic principles and existing technologies were described in this chapter. This chapter does not aim to fully capture all details of Compo language, because it was exhaustively explored and described in [2].

Following chapters go deep into a virtual machine design and implementation.

[13]Component-Based Software Engineering

Chapter 2

Design

This chapter discusses the following design topics: toolchain settings, extraction and final definiton of Compo grammar, design of frontend and virtual machine core details. The virtual machine is implemented in C++ programming language. Details on choice of C++ programming language can be found in 3.1.1.

2.1 Overview

A very basic question *"What is the result of this thesis?"* is answered here.

The result of this thesis is a prototype of the virtual machine for Compo language. The virtual machine is implemented as an AST[14] interpreter, thus the frontend does not produce any bytecode, but AST. The prototype should solve problems like bootstrapping, reflection and inheritance.

2.2 Used tools

Since toolchains and development environments are crucial every-day-life components for any programmer, it is appropriate to devote few words to this part.

2.2.1 Flex and Bison

As described in [8], Flex and Bison are tools designed for writers of compilers and interpreters, although they are also useful for many applications that will interest noncompiler writers. Any application that looks for patterns in its input or has an input or a command language is a good candidate for Flex and Bison. Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance of programs.

There is also a port for Microsoft Windows of these tools.

[14]Abstract Syntax Tree

2.2.2 Perl language

An integral part of developer's skills is the ability to write little helper scripts that make his life easier. Perl language was used to create such scripts. This choice was made because the Perl is a multiplatform and a very powerful language. Perl was also used to simplify the process of typesetting of this thesis.

2.2.3 Doxygen

Doxygen was used to create programmers reference. Following chapter may refer to particular parts of doxygen documentation. Perl script for documentation generation is included in the attachments.

2.2.4 OCLint, CppCheck

OCLint and CppCheck static analyzers were used to improve the code quality. Perl script for generation of static analysis is included in the attachments.

2.2.5 Valgrind

Valgrind is a multipurpose code profiling and memory debugging tool for Linux when on the x86 and, as of version 3, AMD64, architectures. Valgrind is automatically called during the tests and a report is generated into XML[15] file.

Command-line parameters used:

```
– –leak–check=full – –show–reachable=yes – –track–origins=yes
```

2.2.6 Ninja

Ninja is a small build system with a focus on speed. It differs from other build systems in two major respects: it is designed to have its input file generated by a higher-level build system, and it is designed to run builds as fast as possible. Ninja was used because of quite a large amount of code, which was relatively difficult for GCC[16] to build in feasible time. Build times with Ninja was reduced approximately by half. Time was measured on native Arch Linux system, Kernel version 4.4.3, CPU: i7, 1.60GHz.

Release version of Compo vm is built with GCC.

2.2.7 CMake

CMake is multiplatform free software for automated builds in various operating systems. CMake is supported in Microsoft Visual Studio, Apple Xcode and

[15] eXtensible Markup Language
[16] GNU Compiler Collection

Linux make. CMake also integrates Valgrind, Flex, Bison, Boost, all of which are included in this project. Main configuration file is called *CMakeLists.txt*, this file must be preserved. CMake generates a lot of temporary files. So called *out-of-source build* places temporary files into separate directory, usually named *build*. CMake is also integrated with NetBeans IDE[17].

2.3 Used libraries

STL[18] and Boost[9] libraries were heavily utilized during the development process.

2.3.1 STL

Besides some basic data structures such as a vector or a stack, the STL was mainly used for shared pointers implementation. Shared pointers proved to be very helpful element, because they simplified dynamic memory operations.

2.3.2 Boost

Boost library is mainly used for tests. It allows to create various test suites, test cases and even to separate them into independent executable binaries. This project uses only one testing binary (compiled from multiple source files) with a large amount of testing code. For further details on tests, see 4.

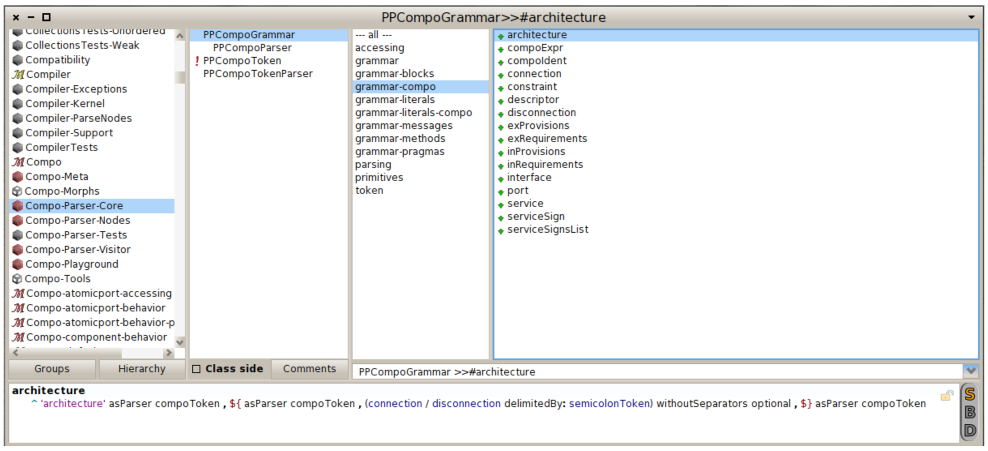
2.4 Grammar extraction

Grammar was extracted from existing implementation in Pharo Smalltalk development environment. Grammar in Pharo is implemented using PetitParser tool. PetitParser is described in [10] as a parsing framework which makes it easy to define parsers with Smalltalk code and to dynamically reuse, compose, transform and extend grammars. Furthermore, PetitParser is not based on tables such as SmaCC and ANTLR. Instead, it uses a combination of four alternative parser methodologies: scanerless parsers, parser combinators, parsing expression grammars and packrat parsers. As such PetitParser is more powerful in what it can parse.

Pharo Smalltalk development environment window is shown in figure 2.1. Window presents *System browser* and its subviews. Subviews are thoroughly described in [11]. Subviews (respectively from left) represents packages, classes, protocols and methods. The bottom view shows Smalltalk code — rule of PetitParser.

[17]Integrated Development Environment

[18]Standard Template Library, standard library for C++ language



```
architecture
    ^ 'architecture' asParser compoToken , ${ asParser compoToken , (connection / disconnection delimitedBy: semicolonToken) withoutSeparators optional , $} asParser compoToken
```

Figure 2.1: Architecture rule in Pharo Smalltalk development environment.

2.4.1 Grammar

Grammar has three sections. First section describes regular expressions for identifier, constant and string literal matching.

Second section is typical procedural grammar. The procedural part is inspired by C grammar, especially in an operator precedence part.

Third section contains Compo specific statements.

For complete grammar refer to appendix A

2.5 Frontend

Generally, frontend of any compiler-like application is composed of a lexical analyzer (scanner), a syntactic analyzer (parser) and a generation of middle- or directly back-end code. A lot of various libraries and tools are available but only few of them are suitable. Following enumeration contains short reasoning about the most well known tools for frontend generation.

- **ANTLR**: Sparse documentation for other languages than Java, supports only LL(*) grammars.
- **Boost Spirit**: Confused description of grammar rules. Grammar rules are described with valid C++ code. For instance, an asterisk symbol[19] must not be placed after an identifier alone, so the *zero or more* occurrences are described with asterisk symbol **before** regular expression insted of **after** regular expression.
- **LLVM**: LLVM contains own Kaleidoscope language, nevertheless, it is suitable primarily for implementing whole language for LLVM backend.

[19] Also known as Kleene star

- **Flex + Bison**: Supports LR grammars as well as C++ code generation.

From previous listing follows, that Flex and Bison combination is the most suitable for this work. Thus, Compo's frontend is built upon Flex scanner, Bison parser and a custom code for AST[20] generation.

Flex scanner is composed of regular expressions to match particular tokens. Scanner is very straightforward in case of Compo. It contains only bunch of keywords and few, more or less complex rules for identifiers, strings, constants and comments matching. Tokens are defined in parser and parser header is included in lexer. Flex file has .l extension.

Bison parser is quite complicated as it contains a lot of custom C++ code and some intricate empty rules for special actions. In fact, Bison parser builds an AST structure for later use. Bison file has .y extension.

Parser code contains various helper structures to hold currently parsed rules. These structures are crucial because of Compo grammar complexity. Some rules need additional actions to be taken.

A bit challenging was to implement and debug stacking contexts of nested compound statements and nested service calls. Bison generates large bunch of C++ (or C) code and it is possible to trace that code in debugger. Since grammar rules have numeric identifiers instead of a name, the code is confusing for quick navigation, etc. Therefore, one must orientate by C++ context. Fortunately, majority of the code was moved into custom *ParserWrapper* class, so the debugging was simplified a little bit.

For thorough description of frontend refer to the chapter 2.

2.5.0.1 Shift-reduce conflicts

Grammar contains two shift-reduce conflicts. First conflict is in the *selection_statement* rule. This particular *selection_statement* conflict is described in [12], section Algorithm, subsection Shift/Reduce Conflicts. This conflict is inevitable and Bison solves this issue by preferring shift before reduction. Second conflict is in the *expression* rule. Bison solves this issue in the same manner as previous conflict.

2.5.1 Abstract Syntax Tree

Abstract syntax tree is a common structure representing the source-code of an application. Various kinds of trees could be found in the real-world compilers and they are covered in [13]. Following listing describes major types:

- **Parse tree**: records the sequence of rules that parser applies as well as the token it matches. It is not very useful for building interpreters and translators.

[20]Abstract Syntax Tree

- **Homogenous AST**: implements an abstract syntax tree using a single node data type and a normalized child list representation. This makes it particularly easy to build external visitors, which rely on a uniform child list for walking.
- **Normalized heterogeneous AST**: implements an abstract syntax tree using more than a single data type but with a normalized child list representation. This pattern makes the most sense when it is necessary to store node-specific data. It is also easier to build external visitor.
- **Irregular heterogeneous AST**: implements an abstract syntax tree using more than a single node data type and with an irregular child list representation[21]. Disadvantage of this approach is more complex visitor implementation.

Compo frontend uses last option due to the fact, that Compo grammar is complicated and with many exceptions. It would be difficult to traverse a child vector in homogeneous AST and search for some particular node. In fact, visitor implementation is large bunch of code and the complexity raises with every new node. But it pays off more than searching and traversing vectors of childs.

Figure 2.2 shows an example of irregular AST for while loop. Figure 2.3 shows an example of irregular AST for descriptor. Note that every AST node has a different type.

2.6 Bootstrapping

Bootstrapping is known as a procedure where a simple computer program initializes a complex computer program. The most well known example of bootstrapping is BIOS[22]. In context of virtual machines, it is understood as a process that produces a minimal functional system (kernel). Virtual machine bootstrapping is described in [14].

To go deep into Compo's bootstrapping mechanisms, it is necessary to introduce concepts of introspection, reflection and reification. Following definitions are taken from [15]

Definition 1. ***Introspection*** *is the ability of a program to observe, and therefore reason about its own state.*

Definition 2. ***Reflection*** *is the ability of a program to modify its own execution state or alter its own interpretation or meaning.*

[21] Instead of uniform list of children, each node data type has specific (named) child fields. This leads to more readable code

[22] Basic Input/Output System

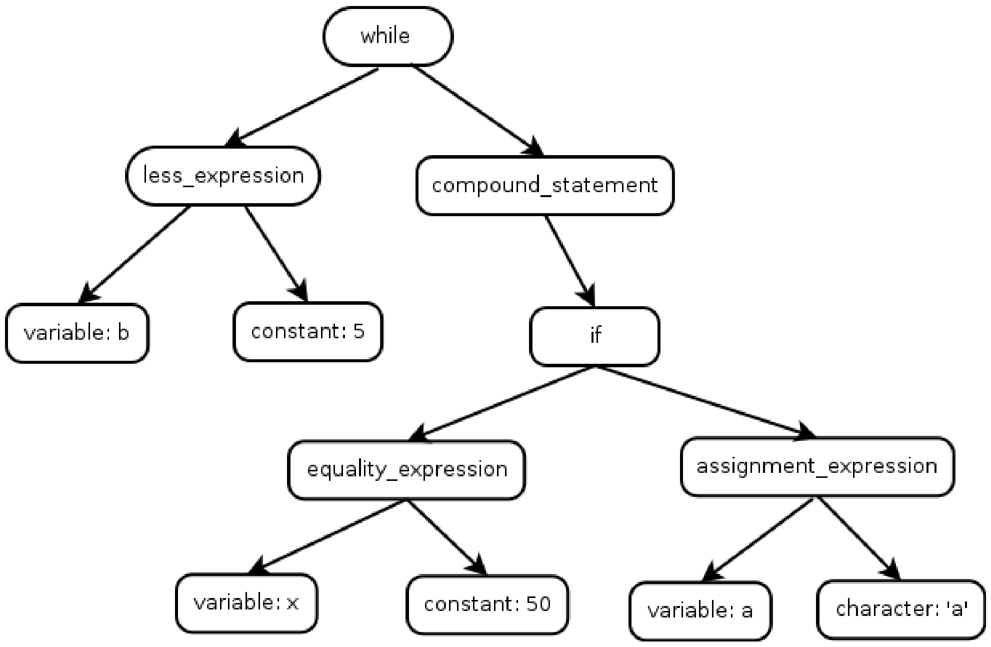


Figure 2.2: Abstract syntax tree for while loop.

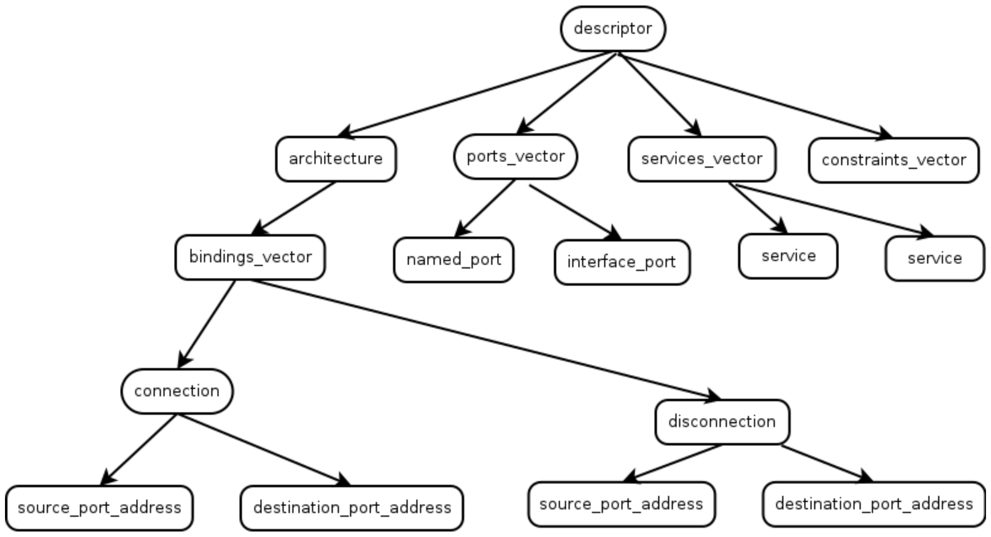


Figure 2.3: Abstract syntax tree for descriptor.

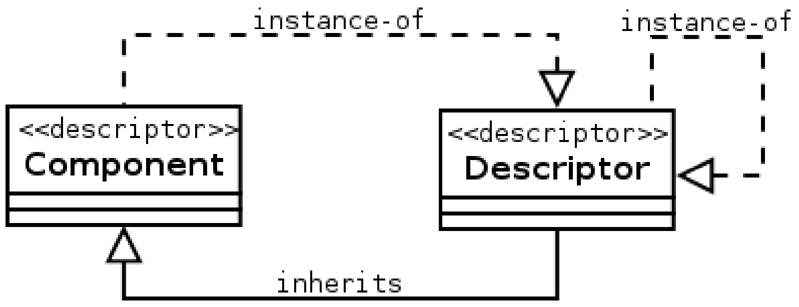


Figure 2.4: Essential part of Compo meta-model: relationship of Component and Descriptor.

Definition 3. *Both of introspection and reflection require mechanism for encoding execution state as data. Providing such an encoding is called **Reflection**.*

There are two kinds of reflection: *structural* and *behavioral*. The structural reflection is considered easier to implement. Languages such as Lisp, Prolog, Smalltalk, and others have included structural reflection for a long time. The structural reflection involves for example addition and alteration services at runtime.

The behavioral reflection touches aspects governing the semantics of programs. The behavioral reflection is not described in this thesis, for further details on behavioral reflection refer to [16].

2.6.0.1 Compo reflection

The structural reflection in Compo is captured by following two definitions:

Definition 4. *Descriptor Component is a basic descriptor and the root of inheritance tree, all descriptors inherit it.*

Definition 5. *Descriptor Descriptor is a sub-descriptor of descriptor Component. It describes descriptors (it is a meta-descriptor) and is the instance of itself.*

For further details refer to [2].

2.6.1 Bootstrapping approaches

A lot of existing approaches were described and implemented. Some of them are described in [14] for Smalltalk virtual machine. Basic idea behind Smalltalk bootstrapping is to obtain existing kernel[23] and load it into the virtual

[23]Minimal system, containing basic language constructs

machine memory. Altough, this introduces the problem of an egg and an chicken: where the first kernel was made? The answer is that the very first Smalltalk virtual machine had to generate kernel by itself.

2.6.2 GNU Smalltalk bootstrapping from scratch

As stated in [14], GNU Smalltalk is the only Smalltalk implementation that is able to recreate a new image from scratch. The GNU Smalltalk virtual machine, written in C, performs this task. The bootstrap function in the virtual machine creates some objects like true, false, nil, the characters, a symbol table, etc. Next, the class and metaclass hierarchy is created. For each class, there is a C struct that stores all its information like its shape, its name and its instance variables. Finally, an entry is added into the global symbol table for each class. Next, the kernel source files are loaded file by file and are executed as a regular Smalltalk execution.

The main advangate of this approach is to produce a clean image. Unfortunately all the process is defined in C as a part of the virtual machine code. Therefore it is tedious to change and reflection can not be utilized at this stage.

2.6.3 Compo Bootstrapping

Bootstrapping from scratch approach was chosen due to the fact that there is no previous existing Compo virtual machine. Thus, no previous kernel can be obtained and loaded into the virtual machine memory.

Like in GNU Smalltalk, there are two stages of bootstrapping.

2.6.3.1 Stage 1

To build Component component and open up reflection capabilities it is necessary to define *primitive* structures, which the Component component is composed from. Definition **6.** is taken from [2]. Definition **7.** was newly invented for the same reason as the primitive port.

Definition 6. *Primitive port behaves like a port but is not a component. Ports declared in the Port descriptor (or its sub-descriptors) are automatically made primitive to avoid infinite regression.*

Definition 7. *Primitive service behaves like a service but is not a component. Services declared in the Service descriptor (or its sub-descriptors) are automatically made primitive to avoid infinite regression.*

First stage of bootstrap builds basic Component component, listed in appendix B, from primitive structures. At this moment of bootstrapping, there is no non-primitive structure available, thus, there is no other option.

Next, all other basic components (Port, Service, PortDescription, etc.) are built from primitive structures. It may look convenient to compose only Component, Service and Port components from primitive structures and others build on top of these. But there is an underlying issue. Port component depends on Interface component, Interface component depends on ServiceSignature component, etc. Thus, when one wants to create Port component, he first has to create Interface and ServiceSignature, but there is no non-primitive Port at the moment. So, for the sakes of clarity, everything in the first stage is built up from primitive structures.

The previous paragraph introduced descriptors, which are not defined in [2], although they are used as inner structures of given descriptors. Newly invented descriptors are listed in appendix B

At this moment, basic components exist and at the same time they provide reflective properties.

2.6.3.2 Stage 2

Second stage of bootstrap builds Descriptor component on the top of previously created components. There is one exception: service *new()* has to be created as primitive and filled with a native code. This code takes all information contained in a descriptor and builds an appropriate component object in the memory.

Second stage also creates System component for basic interaction with real world. System component contains input/output operations, in the future it can also contain networking, filesystem operations, etc.

Second stage bootstrap is a unique interface to the interpreter. Calling second stage bootstrap methods is the only way how to obtain a new component of any type.

2.6.4 Inheritance

Compo inheritance is defined in [2] — it says that Compo supports *only* single inheritance (choice 24 in [2]) and every descriptor inherits directly or transitively (via its parent's descriptor) from Component descriptor (choice 35 in [2]).

Inheritance is one of important features of Compo. An essential inheritance usage is shown in figure 2.4. Some kind of inheritance has to be present during both stages of bootstrapping.

Inheritance in the first stage of bootstrapping is present in a form of hard copy of parent properties to the child component.

Inheritance in the second stage of bootstrapping is done by referencing to the parent and child component. The descriptor always holds the name of the parent descriptor. In the process of the new component creation, a parent

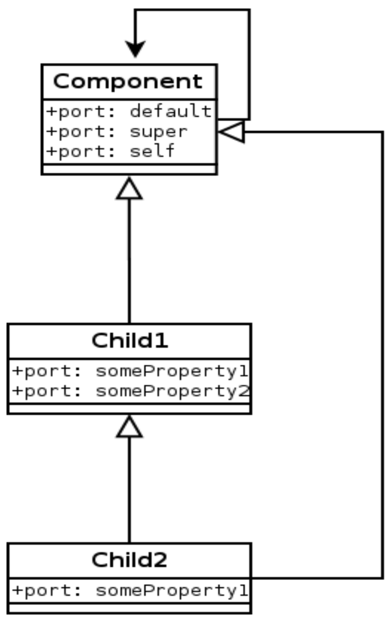


Figure 2.5: Clash of parent components.

component (respectively components) is (respectively are) created first and then connected to the child components.

2.6.5 Inheritance issue

There is an issue with inheritance. Every component is derived from Component component, which posseses *super*, *self* and *default* port. In [2] is stated that the *super* and *self* ports are delegated to the *default* port of a parent component. This definition introduces a clash of connections, because there are two kinds of parents. Refer to the figure 2.5. There is a base Component component and two inherited components. Child1 inherits from Component. Child2 — as stated in subsection 2.6.4 — inherits from Component descriptor. At the same time, there is no support for multiple inheritance, so there is no way how to inherit from Child1. Probably the multiple inheritance is the solution for this issue.

2.7 Interpreter execution

Running of computer programs can be realized in various ways. The straightforward way to execute a program is to rewrite it to a machine code of a given machine and run the code. This is a very complex task for high-level languages. Compo is a high-level language.

Further options are far simpler and involve virtual machines. Three very basic approaches are:

- **Syntax-directed interpreter**: This approach is usually used in simple domain-specific languages, e.g. SQL.
- **Tree-based interpreter**: Tree interpretation is used by dynamic languages like Lisp or Prolog.
- **Bytecode interpreter**: Bytecode interpreter is the most complex way to interpret languages. Standard object-oriented languages like Java go this way.

Intended outcome of this thesis is an AST interpreter. Interpretation is done in a typical way using an endless loop with large *swith-case* commands, where every *case* processes a particular AST node. As there is a requirement to compile the service code during the runtime, an interpreter holds reference to the parser and provides a method for service code parsing.

The interpreter also contains two structures for semantic checking.

First structure is a descriptor table, which holds references to all user-defined descriptors as well as Compo descriptors (Service, PortDescription) to maintain reflective properties such a service and port addition.

Second is service context stack which holds context of the currently executed service and its sub-compound statements.

2.8 Memory

The whole project heavily utilizes shared pointers from STL. Due to the fact that Compo's internal structure is complicated, every component has to be allocated and deallocated in a systematic way. Many circular dependencies occur, thus it is not possible to maintain a correct dynamic memory structure without any memory management.

The memory is implemented as multiple vectors, holding strong references[24] to any kind of inner objects. Each vector holds references to different kind of inner object to distinguish the way the object is managed. Inner objects are components, primitive services and primitive ports. All of the inner objects can have circular dependecies on each other.

Components, primitive ports and primitive services hold weak references to their inner properties.

[24]Strong reference holder is entity, which is responsible for memory deallocation.

Chapter 3

Implementation

3.1 Implementation notes

3.1.1 Choice of programming language

C++ programming language was chosen to implement Compo virtual machine. C++ code benefits mainly from STL shared and weak pointers. Thus, almost no memory management is needed and no *new* or *delete* C++ keywords could be found across the code.

3.1.2 Memory management

As stated in section 2.8, the internal structure of Compo is complicated and circular dependencies are inevitable. So the reasoning about weak pointers is suitable at this place. Every structure mentioned further is managed with a weak pointer or a shared pointer.

3.1.3 Namespaces

The whole project is divided into namespaces to avoid naming clash. The disadvantage of using namespaces is the length of fully qualified names. Therefore, namespace definitions are used. Directory *include/definitions* contains AST definitions, memory objects definitions as well as interpreter entities definitions. There are also STL pointers definitions. The decision to create these definitions was made due to the code complexity and poor readability. Downside of this approach is slowed compilation because of many included files. Namespaces reproduce directory tree hierarchy.

3.2 Grammar implementation

Grammar is implemented by using Flex lexer and Bison parser. The description and examples follow. Only basic examples are shown, for complete listing refer to the attached source code.

3.2.1 Flex scanner

Example of a rule in Flex is shown in listing 3.1. There is an *Identifier* variable which holds a regular expression for identifier matching. Note that identifiers in Compo are allowed to start with upper-case letters. *<INITIAL>*stands for initial state of lexer, therefore *Identifier* and *architecture* have to be matched from initial state as well as *COMMENT* state. It is evident that state *INITIAL* is changed to *COMMENT* with slash and asterisk symbols, everything else is ignored (including newlines) and with another slash and asterisk symbols, *COMMENT* state is changed back to the *INITIAL* state. Identifier is represented by instance of class *CSymbol*, pointer to which is sent through *yylval* variable.

```
Identifier                    [a-zA-Z][a-zA-Z_0-9]*

<INITIAL>{Identifier} {
  yylval = std::make_shared<nodes::procedural::CSymbol>
      (std::string(yytext, yyleng));
  return IDENTIFIER;
}

<INITIAL>"/*"                 {BEGIN(COMMENT);}

<COMMENT>"*/"                 {BEGIN(INITIAL);}

/* Multi-line comments allowed */
<COMMENT>\n                   {}

<COMMENT>.                    {}

<INITIAL>architecture         {return ARCHITECTURE;}
```

Listing 3.1: Flex rules example.

3.2.2 Bison parser

3.2.2.1 Basic rule

Bison parser utilizes tokens received from Flex lexer and builds syntactic rules upon them. Appropriate action is taken while the rule is matched. Usually it is creation of some particular AST[25] node and sending pointer to node into the pseudo-variable $$. Pseudo-variable $$ stands for the semantic value for

[25]Abstract Syntax Tree

the grouping. The lower-case string means a syntactic rule, upper-case means a token. The descriptor rule matches keyword *DESCRIPTOR*, name of the descriptor represented by an *IDENTIFIER*, *inheritance* rule and *compo_expr* rule enclosed in brackets. The only important parts are captured in listing 3.2 for sakes of simplicity (for example shared pointers are omitted).

3.2.2.2 Nested structure rules

Rules for nested structures parsing are very interesting and they required some reasoning. In the first place, it is necessary to clarify, how the Bison parser works. While there is a rule with a related action, the action is taken after the whole rule — with all its sub-rules — is matched. As long as grammar contains nested structures, there is a need to save the context before the rule is matched. There are two solutions for this situation:

1. Place brackets with C++ code before the nested rule. This is also called the "mid-rule"[26].

2. Add an empty rule that contains only brackets with C++ code to the beginning of the nested rule

Second option was selected for the sakes of clarity. First option looks quite tangled, because the second half of the rule may be confused with alternative rule. This rule is shown in listing 3.2. Note the rule *push_context*, which does not have any associated statements on the right side of the rule. An action is taken if this rule is expanded.

[26]`http://www.gnu.org/software/bison/manual/html_node/Using-Mid_002dRule-Actions.html#Using-Mid_002dRule-Actions`

```
compound_statement
: push_context '{' temporaries statement_list '}'
  { $$ = parser->getCurrentCompoundBody();
    parser
     ->setCurrentCompoundBody(parser->popBlock());
  }
| push_context '{' '}'
  { $$ = nullptr; }
;

push_context
:
  { parser->pushBlock
         (parser->getCurrentCompoundBody());
    parser->setCurrentCompoundBody(CCompoundBody());
  }
;
```

Listing 3.2: Bison rule example.

3.2.2.3 Rule for matching service code

To fulfill basic requirement of Compo — to parse service code during runtime — there is one special rule.

While going through parsed text, the parser exactly matches tokens, given by the lexer. Tokens are divided by whitespace characters. It is necessary to do some magic to match the whole service code (with all whitespaces and newlines) at once and preserve the result.

Bison parser thus contains a special rule to switch Flex state. This rule is listed in listing 3.3. Note that C++ code in curly brackets **before** a parser state is executed first, then the rule is expanded. So the state is switched before the token is parsed. After switching the lexer state, the lexer is instructed to match input regardless of whitespaces. Clue for the lexer are opening and closing brackets ('{' respectively '}').

The lexer contains a counter. The counter is incremented with opening bracket and decremented with closing bracket. When the counter is equal zero, the state is switched back to the initial state and a string literal token is returned.

Previous section offered two options for "mid-rule" and the second was chosen. First option was chosen for this issue due to the fact that an action to be taken is only one command and in this simple case it will not be confused with an alternative rule.

```
service_code
: { parser->getLexer()->setServiceState(); }
        STRING_LITERAL
  { $$ = $2; }
;
```

Listing 3.3: Special Bison parser rule to match whole service code.

3.2.2.4 Operator precedence rules

There are a lot of existing solutions of how to implement operator precedence rules in grammar. A brief description with reasoning about which one to choose follows.

- **Edsger Dijkstra's shunting yard algorithm**: First described in [17]. Have to be implemented as a whole algorithm from scratch. Not suitable for simple addition of rules into the grammar.

- **Vaughan Pratt's top down operator precedence method**: First described in [13]. Recursive descent parser. Not suitable for Compo LR grammar.

- **Precedence climbing method**: Described in [18]. Not as efficient as previous methods; form of parser rules; suitable for needs of Compo.

Listing 3.4 shows the practical usage of a precedence climbing method. Listing contains only a short excerpt.

```
primary_expression
  : IDENTIFIER
    { $$ = $1; }
  | CONSTANT
    { $$ = $1; }
  | STRING_LITERAL
    { $$ = $1; }
  ;

multiplicative_expression
  : primary_expression
    { $$ = $1; }
  | multiplicative_expression '*' primary_expression
    { $$ = CMultiplicationExpression($1, $3); }
  | multiplicative_expression '/' primary_expression
    { $$ = CDivisionExpression($1, $3); }
  ;

additive_expression
  : multiplicative_expression
    { $$ = $1; }
  | additive_expression '+' multiplicative_expression
    { $$ = CAdditionExpression($1, $3); }
  | additive_expression '-' multiplicative_expression
    { $$ = CSubtractionExpression($1, $3); }
  ;

assignment_expression
  : logical_or_expression
    { $$ = $1; }
  | primary_expression ASSIGNMENT expression
    { $$ = CAssignmentExpression($1, $3); }

expression
  : assignment_expression
    { $$ = $1; }
  ;
```

Listing 3.4: Example of precedence climbing method.

```
ast ...................... Top directory of AST structures and algorithms
├─ nodes ................................ Namespace for particular nodes
│  ├─ compo ............................. Namespace for COMPO nodes
│  └─ procedural ............ Namespace for ordinary procedural nodes
└─ visitor ............................... Namespace for AST visitors
```

Figure 3.1: AST namespaces.

3.3 Abstract syntax tree structure

This section describes the AST creation and its basic structures. The AST is created while parsing a source code. A lot of C++ code is included in Bison source file to match the particular rule and build a corresponding node.

3.3.1 Namespaces

Figure 3.1 shows the namespaces structure for the AST part. There are two namespaces for AST nodes. A Compo part and a procedural part are separated to avoid confusion and to distinguish interpretation of an ordinary procedural part and a component part. Third namespace contains various visitors.

3.3.2 AST nodes

As stated in chapter 2, the AST is in a form of an irregular heterogenous tree. Thus, every node type has a different inner structure. Listing 3.5 presents simplified class that represents the AST node for Compo descriptor. Note inheriting from *std::enable_shared_from_this<CDescriptor>* and *virtual void accept(...)* method. These features enable implementing of an external visitor design pattern. Constructor is heavily simplified because of length of default parameters. For details refer to the attached code.

Listing 3.5 is C++ representation of AST depicted in figure 2.3.

```
class CDescriptor
: public CDescriptor,
  public enable_shared_from_this<CDescriptor> {
   private:
     shared_ptr<CArchitecture> m_architecture;
     vector<shared_ptr<CPort>> m_ports;
     vector<shared_ptr<CService>> m_services;
     vector<shared_ptr<CConstraint>> m_constraints;

   public:
     CDescriptor(...);
     virtual void accept(
               shared_ptr<CAbstractVisitor> visitor);

     size_t getServicesSize();
     size_t getConstraintsSize();
     size_t getPortsSize();
     shared_ptr<CService> getServiceAt(int index);
     shared_ptr<CConstraint> getConstraintAt(int index);
     shared_ptr<CPort> getPortAt(int index);
     shared_ptr<CPort> getPortByName(string name);
     shared_ptr<CArchitecture> getArchitecture();
     shared_ptr<CService> getServiceByName(string name);
};
```

Listing 3.5: The Descriptor class.

The root of the inheritance tree is class CNode which contains a type of node. An excerpt of the inheritance tree is shown in figure 3.2.

3.3.3 Visitors

As stated in [19], the visitor design pattern comes under the behavioral pattern category. The visitor design pattern represents an operation to be performed on the elements of an object structure. The visitor design pattern defines a new operation without changing the classes of the elements on which it operates.

Figure 3.3 shows an excerpt from AbstractVisitor class.

Compo vm implements three kinds of visitors:

- **SemanticCheckVisitor**: performs a very basic semantic checking.
- **PrintVisitor**: performs printing of the code in a readable form (with proper whitespaces).

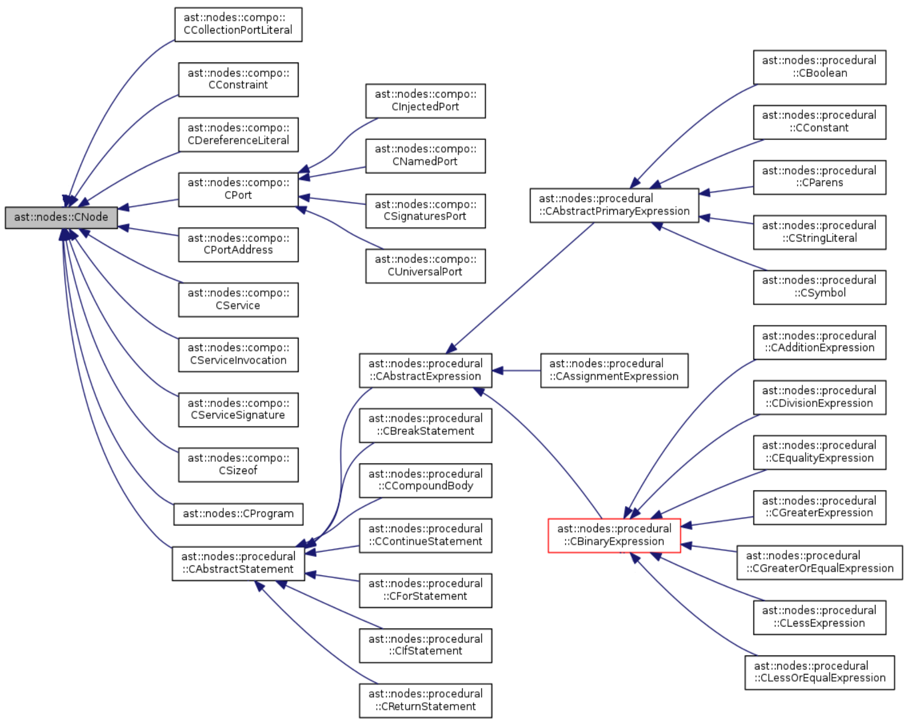

ast::nodes::CNode
ast::nodes::compo::CCollectionPortLiteral
ast::nodes::compo::CConstraint
ast::nodes::compo::CDereferenceLiteral
ast::nodes::compo::CPort
ast::nodes::compo::CPortAddress
ast::nodes::compo::CService
ast::nodes::compo::CServiceInvocation
ast::nodes::compo::CServiceSignature
ast::nodes::compo::CSizeof
ast::nodes::CProgram
ast::nodes::procedural::CAbstractStatement
ast::nodes::compo::CInjectedPort
ast::nodes::compo::CNamedPort
ast::nodes::compo::CSignaturesPort
ast::nodes::compo::CUniversalPort
ast::nodes::procedural::CAbstractExpression
ast::nodes::procedural::CBreakStatement
ast::nodes::procedural::CCompoundBody
ast::nodes::procedural::CContinueStatement
ast::nodes::procedural::CForStatement
ast::nodes::procedural::CIfStatement
ast::nodes::procedural::CReturnStatement
ast::nodes::procedural::CAbstractPrimaryExpression
ast::nodes::procedural::CAssignmentExpression
ast::nodes::procedural::CBinaryExpression
ast::nodes::procedural::CBoolean
ast::nodes::procedural::CConstant
ast::nodes::procedural::CParens
ast::nodes::procedural::CStringLiteral
ast::nodes::procedural::CSymbol
ast::nodes::procedural::CAdditionExpression
ast::nodes::procedural::CDivisionExpression
ast::nodes::procedural::CEqualityExpression
ast::nodes::procedural::CGreaterExpression
ast::nodes::procedural::CGreaterOrEqualExpression
ast::nodes::procedural::CLessExpression
ast::nodes::procedural::CLessOrEqualExpression

Figure 3.2: AST inheritance tree.

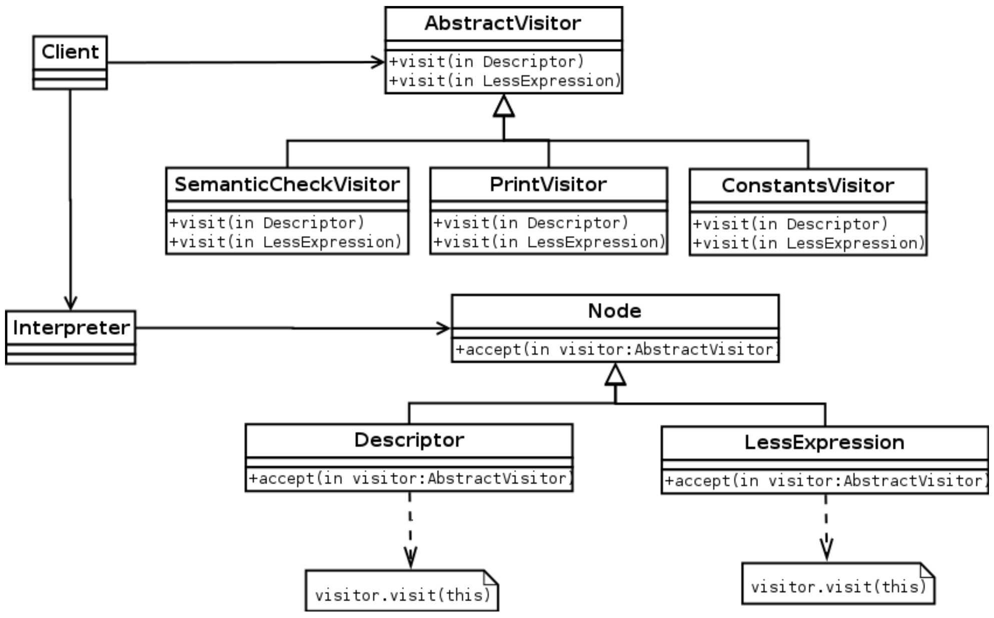


Figure 3.3: Visitor structure.

- **ConstantsVisitor**: searches for constants contained in a given subtree — used by Service component which holds references to constants in its code.

3.4 Memory objects implementation

The VM's memory contains various kinds of objects. Each elementary object is an instance of class CComponent, which represents Compo Component. Component contains references to other components or other non-component objects.

3.4.1 Component

Listing 3.6 presents an excerpt from CComponent class. It is evident that CComponent class holds weak references to its properties. Weak references are necessary because of avoiding circular dependencies.

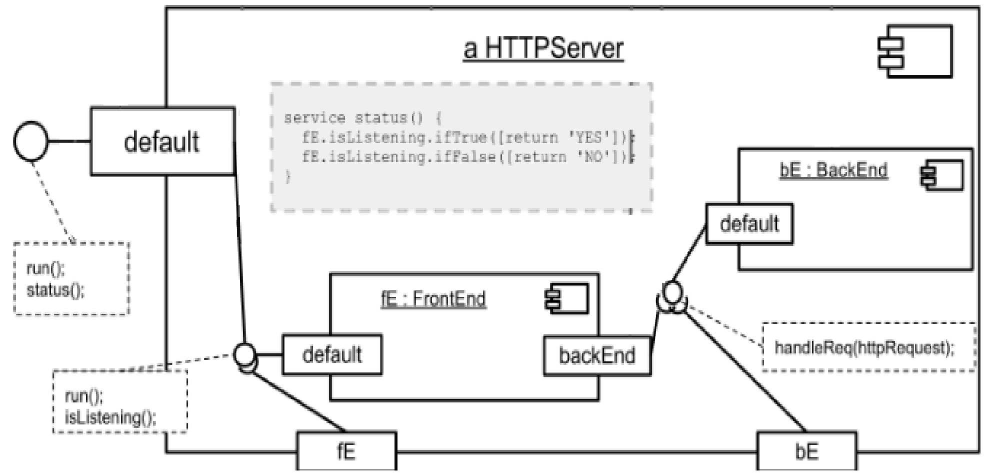
As defined in [2], Component consists of ports (vector of references to instances of class CGeneralPort) and services (vector of references to instances of class CGeneralService). There is also a weak reference to children and a parent. Class CComponent contains many methods for services and ports lookup.

```
class CComponent
: public enable_shared_from_this<CComponent> {
  protected:
    weak_ptr(CComponent) m_parent;
    weak_ptr(CComponent) m_child;
    vector<:weak_ptr(CGeneralPort)> m_ports;
    vector<:weak_ptr(CGeneralService)> m_services;

  public:
    CComponent();
    virtual ~CComponent();
    ...
}
```

Listing 3.6: The Component class.

3.4.1.1 Inheritance

The inheritance is represented by *parentName* port definition in Descriptor descriptor. While creating new instance of a sub-descriptor, all instances of super-descriptors are created first in the proper order. These instances — components — are interconnected with the child and parent references.



```
service status() {
  fE.isListening.ifTrue([return 'YES'])
  fE.isListening.ifFalse([return 'NO'])
}
```

Figure 3.4: Example of complicated component.

3.4.1.2 Component copying

Component copying is a complex task because of tangled Compo inner structure. The natural way to think about copying a component is to copy its inner structure and do not touch outter environment.

Refering to the figure 3.4 taken from [2]. Let's suppose that outter environment holds the reference to the *default* port of HTTPServer and it wants to make a copy of it. As showed in listing 3.6, component holds only references to ports[27]. So the component does not know, whether inner components are connected or not. Therefore, the component itself cannot decide how to interconnect a newly created copy.

Next step is to distinguish components, which are in fact Port components. Port components would be copied in the same manner as the standard component with one exception. The port itself has to decide, wheter it is an internal or an external port and what is its role. Moreover, this information is available at the runtime because everything (including port) is a component.

This implies that a simple copy constructor is not sufficient and extra work must be done to copy the whole inner structure.

The copy constructor can do the job of copying ports, services, parents and children. Extra work lies in duplicating connections, which depend on the runtime information.

[27]References to ports are **not** references to the connected ports, but references to the instances of Port descriptor (primitive ports respectively).

3.4.2 Ports

Compo defines two kinds of ports: primitive ports and instances of Port descriptor. Nevertheless, a component is not aware of primitive ports. Components deal with all ports in the same manner. The class CGeneralPort was invented for this reason. Class CGeneral port provides a uniform interface to the user and internally holds a reference to the real port. The real port is a primitive port (exclusively) or a Port component.

CGeneralPort also provides semantic information about ports such a visibility, role and primitiveness flag. This is useful especially while validating service invocation through a particular port. Interpreter does not have to search for inner structures and simply calls a suitable getter.

3.4.3 Services

As it was described in [2], Compo defines only Service component. But there is a problem of infinite regression while building the Service component, because the Service component also contains service(s). Thus, primitive services were added.

The situation is the same as for ports at this point. CGeneralService class is implemented and it is treated as a Service. The Service component contains a service code in a text form, which allows the programmer to alter a service code dynamically at the runtime.

3.4.4 Primitives

This section describes primitive entities of Compo. Primitive entities are necessary elements to avoid infinite regression.

3.4.4.1 Abstract primitive class

CAbstractPrimitive is a base class to inherit from for primitive entities. Usual Compo component contains a name, e.g. a port name or a service selector, and an owner (respectively context). CAbstractPrimitive class implements these properties with std::string instance variable and weak pointer to owner CComponent object.

3.4.4.2 Primitive port

An excerpt from CPrimitivePort implementation is shown in listing 3.7. CPrimitivePort class contains all properties, according to Port descriptor. *owner* and *name* ports are included in CAbstractPrimitive base class. *connectedPorts*, respectively *delegatedPorts* ports are represented by *m_connectedPorts*, respectively *m_delegatedPorts*. *interfaceDescription* — if any — is represented by *m_connectedServices.*

```
class CPrimitivePort : public CAbstractPrimitive {
 private:
  vector<weak_ptr(CGeneralPort)> m_connectedPorts;
  weak_ptr(CGeneralPort) m_delegatedPort;
  vector<weak_ptr(CGeneralService)> m_connectedServices;
  bool m_isDelegated;
  bool m_isCollection;

 public:
   CPrimitivePort();
   virtual ~CPrimitivePort();
};
```

Listing 3.7: The primitive port class.

3.4.4.3 Primitive service

The primitive service contains only *std::function<...* > callback which executes its code while invoked. The code of a primitive service is filled in during bootstrapping.

```
class CPrimitiveService : public CAbstractPrimitive {
 private:
  function<ptr(CGeneralPort)(const ptr(CComponent)&)>
          m_callback;

 public:
  CPrimitiveService();
  virtual ~CPrimitiveService();
};
```

Listing 3.8: The primitive service class.

3.5 Exceptions

Exceptions provide a way to react to exceptional circumstances (like runtime errors) in programs by transferring control to special functions called handlers.

Compo virtual machine contains many exceptions to distinguish all error states. Elaborated error handling is a foundation for successful debugging and tracing of complex applications. A lot of exception types proved to be handy tool to track unhandled states. No exceptions were thrown during experimenting development stage, because it is not worth elaborating deeply something which is not proven to be right. However, it was a big leap to add exceptions when concepts were verified.

3.5.1 Abstract exception

There is CAbstractException class which is derived from std::exception. CAbstractException is derived from std::exception to coalesce with standard C++ exceptions and to enable to catch the most general exception. CAbstractException class contains a string to hold the message set in the constructor.

3.5.2 Semantic exceptions

Semantic exceptions are thrown when wrong semantic behavior is encountered. Acutally, these exceptions are thrown both in compile time and at runtime. Semantic exceptions comprise redefined/undefined variables or ports, wrong port usage or referencing of wrong AST node.

3.5.3 Runtime exceptions

Runtime exceptions are thrown exclusively at runtime. Runtime exceptions concern calling an unknown service, calling a service with a wrong number of parameters, referencing a wrong port or calling a service on a wrong port. These exceptions were extremely useful during the service invocation implementation.

3.5.4 Execution exceptions

Execution exceptions cover a special case of exceptions. Compo language requires *return* statement. *return* statement cannot be mapped to return statement in C++ but, at the same time, *return* statement has to unroll execution back to the service invocation. Moreover, *return* statement may contain a value. The simple solution to this problem is to throw an exception which will be caught in an invoking code. The return exception contains pointer to the returned value and the invoking code may easily extract returned value.

Same principle is used for continue and break. If continue or break is encountered in a loop code, the rest of the code in the black is skipped and execution whether continues with a new iteration or ends. There are another two special types of exceptions for this purposes.

3.6 Bootstrapping implementation and structures

The bootstrapping code is placed into two separated classes representing two stages of bootstrapping. The second stage possesses a reference to the first stage and the first stage allocates everything through a reference to the

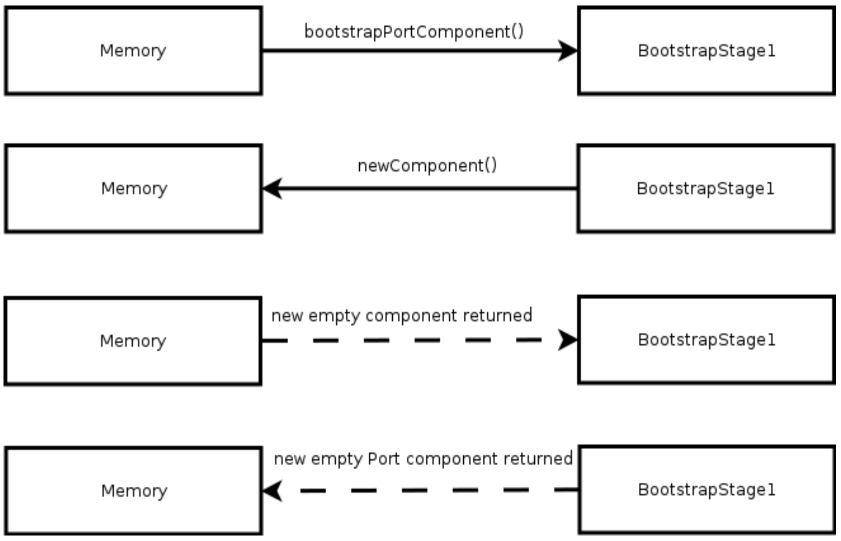


Figure 3.5: Example of communication between first stage bootstrap and memory.

memory. Compo's memory is implemented as a separate class, referencing the first stage of bootstrap. The memory and the first stage of bootstrap benefits from each other. Figure 3.5 shows example of communication between bootstrap and memory objects.

3.6.1 Core modules

Directory *coreModules* contains Compo source codes with basic structures for bootstrapping and reflection. Compo source code of core modules is listed in appendix B. Basic structures help with bootstrapping in a sense of adding ports and bare services. Services are then filled in with the appropriate method and the corresponding callback. As stated in 3.4.4.3, the first stage of bootstrapping contains only primitive services. An original idea was to execute these modules in the second (or maybe third) stage of bootstrapping in Compo interpreter and place them into the descriptor table. Nevertheless, the first stage bootstrapping showed to be a complex task and executing of core modules is left for the future work. First intention was also to have some kind of clue to follow and not to create whole inner structure in C++ code — as it is in a case of GNU Smalltalk.

3.6.2 First stage bootstrapping

First stage bootstrapping is represented by CBoostrapStage1 class.

First stage bootstrapping loads core modules and provides methods to build core components. As there is no inheritance at the first stage of bootstrapping, a bootstrapped component is made of one object without any parent/child component. *bootstrapComponent()* is a method which prepares base Component component, based on information obtained from core modules.

Next there are methods for ports and bare services addition. These methods iterate through the provided AST descriptor (which is parsed from Compo core modules) and create primitive ports and services from provided AST description.

Other components, like Port, PortDescription, Service, etc. are created in the same manner as Component with the exception that their properties are added directly to the Component component without inheritance features.

3.6.3 Second stage bootstrapping

Second stage bootstrapping is represented by CBoostrapStage2 class.

Second stage bootstrapping is created to provide one major method. This major method is *bootstrapDescriptorComponent()* and almost all other methods in CBoostrapStage2 class are only support methods for *bootstrapDescriptorComponent()* method.

bootstrapDescriptorComponent() method takes an AST of the descriptor as a parameter and builds Descriptor component from non-primitive structures. Descriptor component services should be made as non-primitive. However, in order to saving time and also due to the fact that there are unsupported features in Compo interpreter, some of the services are currently primitive.

The most important primitive service of Descriptor component is service *new()*. Service *new()* manufactures Component component from information provided by Descriptor component. In other words, Component component is made from runtime information, contained in Descriptor component. This property directly implies reflection as it is possible to change structure of Descriptor component at runtime and produce a brand new Component.

The second stage bootstrapping also provides methods for obtaining value components.

3.7 Value components

Values in Compo are represented by special components. There is CValue class, which inherits from CComponent — it preserves "is-component" property — and also deletes (at C++ level with keyword *delete*) unnecessary methods. Values also have to be bootstrapped because of one tiny C++ feature.

Every component **must** possesses a default port. Thus, value components **must** also posseses a default port. One may think that it is easy to add a default port while creating a value component. But there is a little hitch. The default port (whether it is primitive or not) holds reference to the owner. So the reference to the owner could be set up in a constructor of a value component (the owner of the default port is just the value component). But this is not possible with shared pointers from STL.

An object which inherits from *std::enable_shared_from_this* must be fully created to pass *shared_from_this()* reference. It implies that an owner reference cannot be set up in a constructor.

Value components creation is included in the second stage bootstrapping.

Compo currently supports:

- **Signed integer**: internally implemented as 8 byte signed int.
- **Boolean**: internally implemented as simple C++ bool data type.
- **String**: internally implemented as std::string.

3.8 System component

System component provides basic services for input and output:

- *print()*: prints provided attribute to the std::cout without newline at the end.
- *println()*: prints provided attribute to the std::cout with newline at the end.
- *readString()*: reads exactly one string (without whitespaces) from std::cin.
- *readLine()*: reads whole line from std::cin.
- *readInt()*: reads integer from std::cin.

There is also an experimental service *getRand()* which returns a pseudo-random number.

3.8.1 Reflection

Current implementation supports all elementary features of a reflection. Although, the interface defined in [2] supports only basic features. Moreover, there was a time constraint to invent a richer interface, that supports operations for real-world application.

Running reflection examples are shown in the testing code. For further details refer to the attached source code or to the chapter 4

3.8.1.1 Reflective descriptor

Since the Descriptor component is a runtime entity, it can be queried and altered at runtime. Descriptor descriptor provides a sufficient interface for a real-world application. There are services for setting the name of the Descriptor, the name of the parent Descriptor, new services, new port descriptions and connections descriptions (from which the architecture of a component is produced). There are of course corresponding getters.

All descriptors are parsed when the interpreter is launched. Then the ASTs of particular descriptors are one by one bootstrapped from the second stage and placed to the descriptors table. Descriptors table is a kind of a global environment available to the programmer through the Compo environment.

3.8.1.2 Reflective ports

The reflection of ports is allowed thanks to the PortDescription descriptor. New instance of PortDescription descriptor can be made, filled and finally joined to the existing descriptor. Second option is to get existing PortDescription from any of the descriptors in the descriptor table and alter its inner properties.

3.8.1.3 Reflective services

Reflection of services is available through the Service descriptor. The service descriptor is also placed in the descriptor table and, thus, is available to the programmer. The major feature is to add a new service source code as a string. Current implementation supports setting of a service selector, a number of parameters and a service code. However, there is a space for future work and new inventions. The current component does not support services like *invokeServiceByName()*, *invokeServiceByArity()*, etc.

3.9 Inheritance

Inheritance in terms of Compo is quite complicated due to the fact that services are tightly coupled with ports. It means that the service invocation depends on a port, which provides a particular service. But that particular port could be provided by a parent component. For example, there is *default* port that is provided by a root component. *default* port provides **all** services. It follows that a parent port has to manage child components services. Another issue is the fact, that any component is handled by its default port. Thus, if one holds reference to a component, i.e. its default port, and tries to invoke some of sub-components services, the interpreter has to look for a service both in upward and downward direction.

Service lookup mechanism, described at [2], algorithm 4, is slightly modified. Modification lies in addition of downward lookup.

3.10 Interpreter implementation

The implementation of an interpreter is quite straightforward with the exception of services invocation. The interpreter contains a long switch-case, where cases represent particular AST nodes. A special action is executed for every single AST node. Every node is processed in a particular method.

Interpreter is represented by CInterpreter class. CInterpreter class contains references to the parser, the second stage bootstrap, the descriptor table and the context stack.

3.10.1 Semantic structures

Semantic structures provide an interface to manage local variables and ports in the current service context.

3.10.1.1 Descriptor table

The most important semantic structure is of course the descriptor table, which contains references to existing Descriptor components. The descriptor table is represented by CDescriptorTable class. Inner representation of the table is implemented with plain vector of references. One may think that hash table of name to reference is more suitable. Nevertheless, reflective properties must be taken into account. The descriptor is uniquely described by the *name* port, which is connected to the String component. So the hash table could contain tuples of String components and references to descriptors. However, there is a risk of errorneous situation, when no String component is connected, then there is no way to determine the descriptor.

It is probably better to have an unaccessible descriptor without any name in the descriptor table than invalid state of the descriptor table. This could be a subject of future work.

3.10.1.2 Context stack

Since the interpreter supports services invocation, some kind of stack structure is necessary. Stack is an elegant and comfortable way to manage nested service invocations. There are two kinds of stack. The first kind is represented by STL stack, which contains references to the CContext class.

CContext class contains a vector from STL that holds references to another class: CVariablesTable. Vector inside of CContext is treated like a stack. Vector is used because of ability to iterate the content and search for a particular element. Another stacking structure is implemented to enable define new variables inside compound statements. Example of usage such a feature is shown in listing 3.9.

```
service test() {
        |i var1 var2|
        for (i := 0; i < var; i := i + 1) {
                |a b|
                a := System.readInt();
                b := 5;
                if (a < b) {
                        |x y|
                        ...
                }
        }
}
```

Listing 3.9: Compound statement context.

CContext class also contains a reference to the current component, respectively to the owner component of the currently executed service. The owner component contains various ports that have to be accessible from inside of the component (for example the self port). While service accesses some of internal ports, CContext searches for the reference to the port like it was ordinary variable. Thus, variables and ports are the same thing from the interpreter point of view.

Ports and variables are tightly linked with the assignment operator. Standard programming languages implement a variable (more or less) as a place in the memory. Compo variable is rather a port than a place in the memory. This is due to the fact that a communication element for Compo component is not its place in the memory (in other words: reference to the component itself), but its default port. Thus, the local variable references the default port of the connected component.

Assignment is defined in [2] as making a copy of the assigned element. Component copying is described in 3.4.1.2 as a complex task, so the current implementation of assignment assigns default port to the variable. Only value components are physically duplicated because of their simple inner structure.

3.10.2 Service invocation

The service invocation depends on the type of invoked service. A C++ callback is directly invoked in case of a primitive service. Situation is a little complicated in case of Service component. Argument passing is defined in [2] as connecting default ports of arguments to the *args* port of context component.

3.10.2.1 Primitive service invocation

As described in 3.4.4.3, the primitive service contains in fact only a name, an owner and a callback. The callback is a pointer to a function which returns reference to a port and takes reference to a component as a parameter. Inner implementation is up to the designer of the bootstrapping. Primitive services are created as C++ lambda functions during bootstraping.

Reference to the component is a context component from which the service is invoked.

The most common use case of a primitive service is to perform some simple operation. Operation may handle connected arguments, return result, perform some operation on context component or both. Arguments are acquired through the context component from an argument of a callback.

Special cases of primitive services are services *execute()* and *new()*.

Service *execute()* determines if arguments port is delegated to another port and checks if a number of arguments matches to its prototype, which is connected to the *serviceSign* port. Then the service code is executed. The execution is made through the call of the *execServiceCode()* from CInterpreter class. Service *execute()* is created as C++ lambda function during the first stage of bootstrapping. Instance of CBootstrap1 class holds reference to the instance of CInterpreter class. Reference to the instance of CInterpreter class is passed as captured variable to the lambda function.

This concept allows the component to hold only reference to a general callback without any information about internals of the callback. The primitive service, when invoked, only calls its callback with context (reference to which is contained in the base CAbstractPrimitive class) as a parameter and returns a result.

Primitive service *new()* is an important part of Descriptor descriptor. Primitive service *new()* produces new instances of a descriptor based on the runtime information saved in the descriptor. Primitive service *new()* is filled during the second stage of bootstrapping. The most complex part is to connect newly produced ports together correspondingly to the content of architecture-Definition. Realizing connections is also complicated with the requirement to build new components directly within the connection. Example of such a connection is shown in listing 3.10.

```
connect fE to default@(FrontEnd.new());
```

Listing 3.10: Example of connection with component instantiation.

3.10.2.2 Service component invocation

The service component follows definition from [2]. The service component contains a code and a service signature. These properties are referenced in *execute()* service. Thus, Service component invocation is made through *execute()* service, described in the previous chapter.

The service component also provides *tempsN* and *tempsV* collection ports. These ports are connected to the variable names, respectively values from the service code.

3.11 Summary

The code contains approximately 14 000 lines of a code. Lines of code was computed with *cloc* utility, which counts only lines with the code — no whitespaces or comments. Simple Perl script for lines counting with non-C++ sources exclusion is included in attachements.

Git repository contains 383 commits.

The code quality was constantly checked with *CppCheck* and *OCLint* static code analyzers.

Chapter 4

Tests

This chapter describes testing of the implemented virtual machine. Tests include unit testing with a code coverage report, real-world examples and the Valgrind memory check report.

4.1 Unit tests

Unit testing is a necessary element in software development. Thanks to the unit tests, developers are able to work on different parts of the system any without worries about hidden inconsistencies of a previously developed code. Unit tests can also serve as a reference of code usage. TDD[28] is closely linked to unit tests. It is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or a new function. Then produces the minimum amount of a code to pass that test, and finally refactors the new code to acceptable standards. TDD was invented by Kent Beck.

4.1.1 Unit testing framework

A lot of unit testing frameworks for C++ are available. Following listing presents the most well known frameworks:

- **Google Test**: Unit test library based on the xUnit architecture, can be compiled for variety of POSIX and Windows platforms.
- **UnitTest++**: Lightweight unit testing framework. It was designed to do a test-driven development on a wide variety of platforms.

[28]Test Driven Development

- **Boost Test**: Provides a matched set of components for writing test programs, organizing tests in to simple test cases and test suites, and controlling their runtime execution.

Boost Test framework was chosen because of its modularity and wide options. Once the Boost is added into the CMake tool, one can simply utilize another Boost library. Additional benefit is the possibility of creation of a separate binary with a test code only.

4.1.2 Test cases and test suites examples

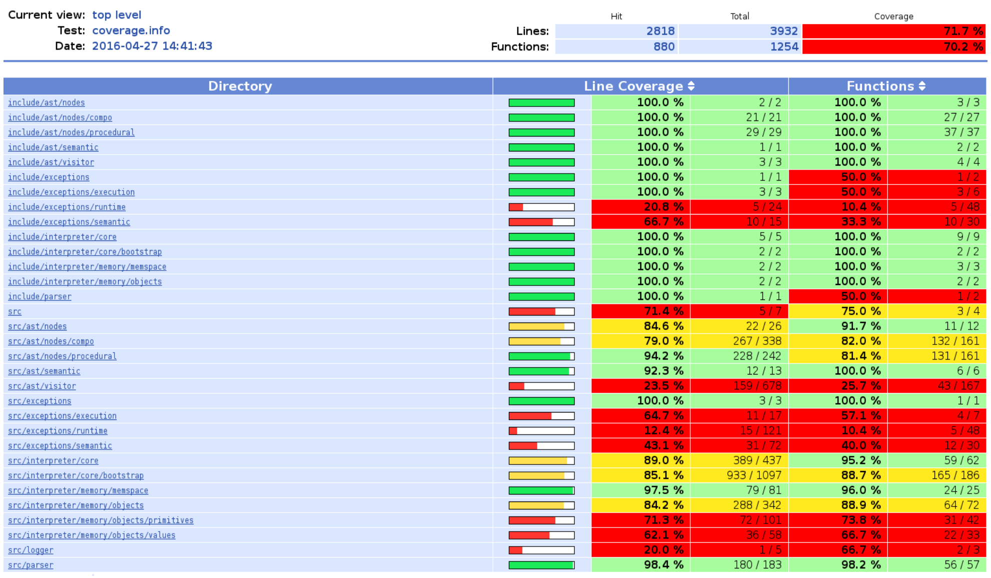
Listing 4.1 presents an excerpt from one of the test suites. This particular test case tests parsing of a service and its parameters. There are some Boost check macros and also custom macros named with prefix *TEST_*. Instance of parser is global for the whole test suite, thus, parser initialization is not present in this excerpt (it is about two lines of a code). Note dynamic pointer casting. This breakneck syntax is present due to the usage of shared pointers.

```
BOOST_AUTO_TEST_CASE(compoServiceParams) {
    stringstream input;
    input.str(
    "descriptor test {\
        service noparams() {}\
        service oneparam(param1) {}\
    }");
    parser.parse(input);

    CDescriptor descriptor = parser.getRootNodeAt(0);
    TEST_DESCRIPTOR(descriptor, "test", "", 4);

    CService service = descriptor->getBodyNodeAt(0);
    TEST_SERVICE(service, "noparams", 0, 0, 0);

    service = descriptor->getBodyNodeAt(1);
    TEST_SERVICE(service, "oneparam", 1, 0, 0);
    BOOST_CHECK_EQUAL("param1",
            service->getParamAt(0)->getStringValue());
}
```

Listing 4.1: Test case example.

4.1.3 Code coverage statistics

Figure 4.1 shows code coverage statistics. Code coverage is not the only one software quality metric. Nevertheless, it can significantly contribute to



Current view: top level
Test: coverage.info
Date: 2016-04-27 14:41:43

	Hit	Total	Coverage
Lines:	2818	3932	71.7 %
Functions:	880	1254	70.2 %

Directory	Line Coverage		Functions	
include/ast/nodes	100.0 %	2 / 2	100.0 %	3 / 3
include/ast/nodes/compo	100.0 %	21 / 21	100.0 %	27 / 27
include/ast/nodes/procedural	100.0 %	29 / 29	100.0 %	37 / 37
include/ast/semantic	100.0 %	1 / 1	100.0 %	2 / 2
include/ast/visitor	100.0 %	3 / 3	100.0 %	4 / 4
include/exceptions	100.0 %	1 / 1	50.0 %	1 / 2
include/exceptions/execution	100.0 %	3 / 3	50.0 %	3 / 6
include/exceptions/runtime	20.8 %	5 / 24	10.4 %	5 / 48
include/exceptions/semantic	66.7 %	10 / 15	33.3 %	10 / 30
include/interpreter/core	100.0 %	5 / 5	100.0 %	9 / 9
include/interpreter/core/bootstrap	100.0 %	2 / 2	100.0 %	2 / 2
include/interpreter/memory/memspace	100.0 %	2 / 2	100.0 %	3 / 3
include/interpreter/memory/objects	100.0 %	2 / 2	100.0 %	2 / 2
include/parser	100.0 %	1 / 1	50.0 %	1 / 2
src	71.4 %	5 / 7	75.0 %	3 / 4
src/ast/nodes	84.6 %	22 / 26	91.7 %	11 / 12
src/ast/nodes/compo	79.0 %	267 / 338	82.0 %	132 / 161
src/ast/nodes/procedural	94.2 %	228 / 242	81.4 %	131 / 161
src/ast/semantic	92.3 %	12 / 13	100.0 %	6 / 6
src/ast/visitor	23.5 %	159 / 678	25.7 %	43 / 167
src/exceptions	100.0 %	3 / 3	100.0 %	1 / 1
src/exceptions/execution	64.7 %	11 / 17	57.1 %	4 / 7
src/exceptions/runtime	12.4 %	15 / 121	10.4 %	5 / 48
src/exceptions/semantic	43.1 %	31 / 72	40.0 %	12 / 30
src/interpreter/core	89.0 %	389 / 437	95.2 %	59 / 62
src/interpreter/core/bootstrap	85.1 %	933 / 1097	88.7 %	165 / 186
src/interpreter/memory/memspace	97.5 %	79 / 81	96.0 %	24 / 25
src/interpreter/memory/objects	84.2 %	288 / 342	88.9 %	64 / 72
src/interpreter/memory/objects/primitives	71.3 %	72 / 101	73.8 %	31 / 42
src/interpreter/memory/objects/values	62.1 %	36 / 58	66.7 %	22 / 33
src/logger	20.0 %	1 / 5	66.7 %	2 / 3
src/parser	98.4 %	180 / 183	98.2 %	56 / 57

Figure 4.1: Code coverage report.

the code quality. Code coverage statistic is generated using GCC parameter *coverage* and web report generation tool *lcov*.

Figure 4.1 shows approximately 70% code coverage which is acceptable result.

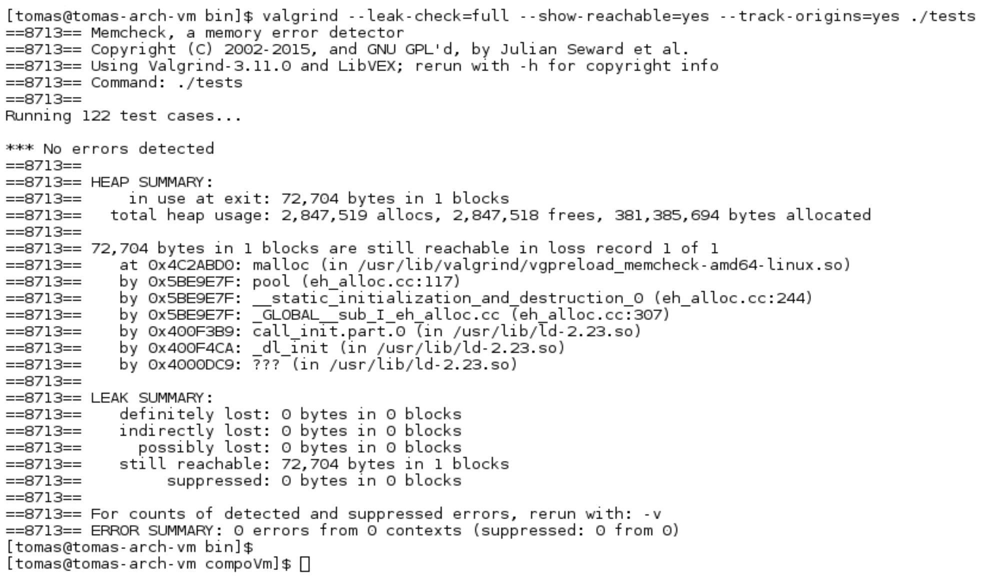
4.2 Real-world examples

Real-world examples are shown in directory *test/compoTests*. They include:

- **helloWorld.cp**: Every programming language "must-have". Prints "Hello world!" to the output.
- **calculator.cp**: Service invocation and procedural statements example. Performs basic arithmetic operations, power and factorial.
- **cars.cp**: Inheritance and ports connections example. Presents service specialization, service addition, ports addition and service lookup mechanism.
- **reflection.cp**: Basic reflection example. Asks user to enter a service code, then executes it and prints the result.

4.3 Valgrind report

The Valgrind check is built within CTest utility of CMake build tool. Valgrind command is included in *CMakeList.txt* file and runs testing binary. Since the



```
[tomas@tomas-arch-vm bin]$ valgrind --leak-check=full --show-reachable=yes --track-origins=yes ./tests
==8713== Memcheck, a memory error detector
==8713== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8713== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8713== Command: ./tests
==8713==
Running 122 test cases...

*** No errors detected
==8713==
==8713== HEAP SUMMARY:
==8713==     in use at exit: 72,704 bytes in 1 blocks
==8713==   total heap usage: 2,847,519 allocs, 2,847,518 frees, 381,385,694 bytes allocated
==8713==
==8713== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==8713==    at 0x4C2ABD0: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==8713==    by 0x5BE9E7F: pool (eh_alloc.cc:117)
==8713==    by 0x5BE9E7F: __static_initialization_and_destruction_0 (eh_alloc.cc:244)
==8713==    by 0x5BE9E7F: _GLOBAL__sub_I_eh_alloc.cc (eh_alloc.cc:307)
==8713==    by 0x400F3B9: call_init.part.0 (in /usr/lib/ld-2.23.so)
==8713==    by 0x400F4CA: _dl_init (in /usr/lib/ld-2.23.so)
==8713==    by 0x4000DC9: ??? (in /usr/lib/ld-2.23.so)
==8713==
==8713== LEAK SUMMARY:
==8713==    definitely lost: 0 bytes in 0 blocks
==8713==    indirectly lost: 0 bytes in 0 blocks
==8713==      possibly lost: 0 bytes in 0 blocks
==8713==    still reachable: 72,704 bytes in 1 blocks
==8713==         suppressed: 0 bytes in 0 blocks
==8713==
==8713== For counts of detected and suppressed errors, rerun with: -v
==8713== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
[tomas@tomas-arch-vm bin]$
[tomas@tomas-arch-vm compoVm]$ 
```

Figure 4.2: Valgrind report.

code coverage concerns about 70% of the code, Valgrind result can be also accepted as successful.

Figure 4.2 shows that there is no serious memory error. Message *still reachable* is Valgrind notation for the memory that was not freed but some reference to the memory exists. As noted in [20], this "error" is usually present while using some external library — like STL. External libraries utilizes their own memory allocators and the memory is kept for later re-use.

4.4 Documentation

Code documentation is generated with doxygen and generating script is included in the attachement. Code documentation describes basic usage of implemented classes.

4.5 Tests summary

All of the mentioned reports and statistics can be reproduced by running generation script. Details on building, testing and reports generating are mentioned in **readme.txt** file.

Chapter 5

Future work

There is a lot of work for successors to make this project a real-world application. Major issues that should be solved are pointed out in the following listing:

- Returning collection port.
- Copying of inner component structure.
- Garbage collection.
- Think out and implement third stage of bootstrapping.

Conclusion

The goal of this thesis was to get familiar with the Compo language, then design and implement front-end with parallel Master's thesis author and finally design and implement a virtual machine for Compo. Compo's virtual machine should be able to evaluate basic language constructs.

The first task was to explore component-oriented paradigm, reflection, search for existing approaches and lay out basic sub-goals. The main source of inspiration was [2] as well as various books, articles and documentations. The analysis chapter of this thesis describes the current state of component-base software engineering.

Next sub-goal was to extract Compo grammar and implement front-end. Grammar extraction was based on prototype implementation in Pharo Smalltalk. Front-end implementation was then quite a straightforward task.

Last and the most difficult part was to design and implement bootstrapping and reflection.

Implementation was done in C++ programming language, using Boost and STL library and Flex & Bison frontend generator tools. The project contains built-in unit tests, helper scripts for building reports and documentation and a few real-world examples of implemented concepts.

Testing was an indispensable process to verify basic code functionality and it is also used as Compo usage reference. The testing code contains approximately 120 test cases and 70% code coverage which could be considered as sufficient.

There are also a few complex examples of Compo usage to demonstrate overall functionality. These examples can be served to a built binary, which asks for the appropriate input and returns results.

The resulting code is definitely not bulletproof. This project can be considered as a first try to implement a pure component-based virtual machine. This project contains some of uncommon concepts and a few humble author's ideas and inventions.

As the most interesting author's contribution can be mentioned mainly a callback structure of primitive services. The primitive service itself is a very simple structure that contains a complex callback. The callback is not managed by the primitive service. The primitive service only invokes the callback. The concept of general services/ports is also quite important. The concept simplifies the management of primitive and non-primitive structures. There was also an effort made to enhance given base-descriptors from [2]. Extended descriptors are shown in appendix B.

However, this thesis is a solid foundation for the future implementation. Successors may study mainly the bootstrapping features and build a better code or think out new practises.

Probably the biggest challenge was to think out and then implement bootstrapping and reflection. The reflection is quite common these days for common languages like Java, Smalltalk, etc. and it opens the door for a lot of new concepts and approaches.

This thesis is primarily based on dissertation thesis of Ing. Petr Špaček, Ph.D. [2] and contained concepts. The dissertation thesis comes up with new approaches in software engineering, which could be widely used in the future.

Bibliography

[1] TIOBE Index for April 2016. 2016. Available from: `http://www.tiobe.com/tiobe_index`

[2] Špaček, P. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language.* Dissertation thesis, Académie De Montpellier, 2013.

[3] Szypersky, C.; Gruntz, D.; Murer, S. *Component Software: Beyond Object-Oriented Programming (2nd Edition).* Addison-Wesley Professional, 2002, ISBN 0201745720, 624 pp.

[4] Aldrich, J.; Chambers, C.; Notkin, D. ArchJava: Connecting Software Architecture to Implementation. *ACM*, 2002.

[5] Henning, M.; Vinoski, S. *Advanced CORBA Programming with C++.* Addison-Wesley Professional, 1999, ISBN 0201379279, 1120 pp.

[6] Fabresse, L.; Dony, C.; Huchard, M. Foundations of a simple and unified component-oriented language. 2008.

[7] Seco, J. C.; Silva, R.; Piriquit, M. ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration. 2008.

[8] Levine, J. R. *Flex & Bison.* O'Reilly Media, Inc., 2009, ISBN 0596155972, 294 pp.

[9] Boost documentation. Available from: `http://www.boost.org/doc/`

[10] Bergel, A.; Cassou, D.; Ducasse, S.; et al. *Deep into Pharo.* 2013, 420 pp.

[11] Black, A. P.; Ducasse, S.; Nierstrasz, O. *Pharo by Example.* Square Brackets Associates, Switzerland, 2009.

[12] GNU Bison manual. Available from: http://www.gnu.org/software/bison/manual/html_node/index.html

[13] Pratt, V. R. Top down operator precedence. 1973.

[14] Casaccio, G.; Ducasse, S.; Fabresse, L.; et al. Bootstrapping a Smalltalk. 2011.

[15] Rivard, F. Smalltalk: a Reflective Language.

[16] Malenfant, J.; Jacques, M.; Demers, F. A Tutorial on Behavioral Reflection and its Implementation.

[17] Dijkstra, E. W. ALGOL-60 Translation. 1961.

[18] Norvell, T. Parsing Expressions by Recursive Descent. Available from: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#climbing

[19] Gamma, E.; Helm, R.; Johnson, R.; et al. *Design patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994, ISBN 0201633612, 395 pp.

[20] Valgrind documentation. Available from: http://valgrind.org/docs/manual/faq.html#faq.reports

Appendix A

Grammar

1 Lexical rules for primary expressions

⟨*identifier*⟩ ::= [a-zA-Z][a-zA-Z_0-9]

⟨*constant*⟩ ::= [+-]?[0-9]+

⟨*string_literal*⟩ ::= ' .+ '

2 Procedural part

⟨*literal*⟩ ::= ⟨*identifier*⟩
  | ⟨*constant*⟩
  | ⟨*string_literal*⟩
  | '**true**'
  | '**false**'

⟨*primary_expression*⟩ ::= ⟨*identifier*⟩
  | ⟨*literal*⟩
  | '(' ⟨*expression*⟩ ')'
  | ⟨*service_invocation*⟩
  | ⟨*connection*⟩
  | ⟨*disconnection*⟩
  | ⟨*delegation*⟩
  | '**sizeof**' '(' ⟨*identifier*⟩ ')'

⟨*multiplicative_expression*⟩ ::= ⟨*primary_expression*⟩
  | ⟨*multiplicative_expression*⟩ '*' ⟨*primary_expression*⟩
  | ⟨*multiplicative_expression*⟩ '/' ⟨*primary_expression*⟩

```
⟨additive_expression⟩ ::= ⟨multiplicative_expression⟩
  |  ⟨additive_expression⟩ '+' ⟨multiplicative_expression⟩
  |  ⟨additive_expression⟩ '-' ⟨multiplicative_expression⟩

⟨relational_expression⟩ ::= ⟨additive_expression⟩
  |  ⟨relational_expression⟩ '<' ⟨additive_expression⟩
  |  ⟨relational_expression⟩ '>' ⟨additive_expression⟩
  |  ⟨relational_expression⟩ '<=' ⟨additive_expression⟩
  |  ⟨relational_expression⟩ '>=' ⟨additive_expression⟩

⟨equality_expression⟩ ::= ⟨relational_expression⟩
  |  ⟨equality_expression⟩ '==' ⟨additive_expression⟩
  |  ⟨equality_expression⟩ '!=' ⟨additive_expression⟩

⟨logical_and_expression⟩ ::= ⟨equality_expression⟩
  |  ⟨logical_and_expression⟩ '&&' ⟨equality_expression⟩

⟨logical_or_expression⟩ ::= ⟨logical_and_expression⟩
  |  ⟨logical_or_expression⟩ '||' ⟨logical_and_expression⟩

⟨assignment_expression⟩ ::= ⟨logical_or_expression⟩
  |  ⟨primary_expression⟩ ':=' ⟨expression⟩

⟨expression⟩ ::= ⟨assignment_expression⟩
  |  ⟨expression⟩ ',' ⟨assignment_expression⟩

⟨statement⟩ ::= ⟨expression_statement⟩
  |  ⟨selection_statement⟩
  |  ⟨iteration_statement⟩
  |  ⟨jump_statement⟩
  |  ⟨compound_statement⟩

⟨compound_statement⟩ ::= '{' ⟨temporaries⟩ ⟨statement_list⟩ '}'
  |  '{' '}'

⟨temporaries⟩ ::= '|' ⟨temporaries_list⟩ '|'
  |  ⟨empty⟩

⟨temporaries_list⟩ ::= ⟨identifier⟩
  |  ⟨temporaries_list⟩ ⟨identifier⟩

⟨statement_list⟩ ::= ⟨statement⟩
  |  ⟨statement_list⟩ ⟨statement⟩
```

⟨*expression_statement*⟩ ::= '**;**'
  | ⟨*expression*⟩ '**;**'

⟨*selection_statement*⟩ ::= '**if**' '**(**' ⟨*expression*⟩ '**)**' ⟨*statement*⟩
  | '**if**' '**(**' ⟨*expression*⟩ '**)**' ⟨*statement*⟩ '**else**' ⟨*statement*⟩

⟨*iteration_statement*⟩ ::= '**while**' '**(**' ⟨*expression*⟩ '**)**' ⟨*statement*⟩
  | '**for**' '**(**' ⟨*assignment_expression*⟩ '**;**' ⟨*expression_statement*⟩ ⟨*expression*⟩
   '**)**' ⟨*statement*⟩

⟨*jump_statement*⟩ ::= '**continue**' '**;**'
  | '**break**' '**;**'
  | '**return**' ⟨*expression*⟩ '**;**'

3 COMPO part

⟨*start*⟩ ::= ⟨*descriptor_interface*⟩
  | ⟨*service_body*⟩

⟨*service_body*⟩ ::= ⟨*compound_statement*⟩

⟨*descriptor_interface*⟩ ::= ⟨*descriptor_interface*⟩ ⟨*descriptor*⟩
  | ⟨*descriptor_interface*⟩ ⟨*interface*⟩
  | ⟨*empty*⟩

⟨*descriptor*⟩ ::= '**descriptor**' ⟨*identifier*⟩ ⟨*inheritance*⟩ '**{**' ⟨*compo_expressions*⟩
   '**}**'

⟨*interface*⟩ ::= '**interface**' ⟨*identifier*⟩ ⟨*inheritance*⟩ ⟨*service_signatures_list*⟩

⟨*inheritance*⟩ ::= '**extends**' ⟨*identifier*⟩
  | ⟨*empty*⟩

⟨*compo_expressions*⟩ ::= ⟨*compo_expression*⟩ ⟨*compo_expressions*⟩
  | ⟨*empty*⟩

⟨*compo_expression*⟩ ::= ⟨*provision_requirement*⟩
  | ⟨*constraint*⟩
  | ⟨*service*⟩
  | ⟨*architecture*⟩

⟨*provision_requirement*⟩ ::= ⟨*visibility*⟩ ⟨*role*⟩ ⟨*provision_requirement_signature*⟩

⟨*visibility*⟩ ::= '**externally**'
  | '**internally**'
  | ⟨*empty*⟩

⟨*role*⟩ ::= '**provides**'
  | '**requires**'

⟨*provision_requirement_signature*⟩ ::= '**{**' ⟨*ports*⟩ '**}**'

⟨*ports*⟩ ::= ⟨*port*⟩ '**;**'
  | ⟨*port*⟩ '**;**' ⟨*ports*⟩

⟨*port*⟩ ::= ⟨*atomic*⟩ ⟨*port_name*⟩ ⟨*collecting*⟩ '**:**' ⟨*port_signature*⟩ ⟨*of_kind*⟩

⟨*atomic*⟩ ::= '**atomic**'
  | ⟨*empty*⟩

⟨*port_name*⟩ ::= ⟨*identifier*⟩

⟨*collecting*⟩ ::= '[' ']'
  | ⟨*empty*⟩

⟨*port_signature*⟩ ::= ⟨*identifier*⟩
  | '*****'
  | ⟨*service_signatures_list*⟩

⟨*of_kind*⟩ ::= '**ofKind**' ⟨*identifier*⟩
  | ⟨*empty*⟩

⟨*service*⟩ ::= '**service**' ⟨*service_signature*⟩ ⟨*service_code*⟩

⟨*service_code*⟩ ::= ⟨*string_literal*⟩

⟨*service_signature*⟩ ::= ⟨*identifier*⟩ '**(**' ⟨*service_params*⟩ '**)**'

⟨*service_signature_call*⟩ ::= ⟨*identifier*⟩ '**(**' ⟨*service_runtime_params*⟩ '**)**'

⟨*service_signatures*⟩ ::= ⟨*service_signature*⟩
  | ⟨*service_signature*⟩ '**;**' ⟨*service_signatures*⟩
  | ⟨*empty*⟩

⟨*service_signatures_list*⟩ ::= '**{**' ⟨*service_signatures*⟩ '**}**'

⟨*service_params*⟩ ::= ⟨*parameter*⟩
  | ⟨*parameter*⟩ '**,**' ⟨*service_params*⟩
  | ⟨*empty*⟩

⟨*parameter*⟩ ::= ⟨*identifier*⟩

⟨*service_runtime_params*⟩ ::= ⟨*parameter_runtime*⟩
  | ⟨*parameter_runtime*⟩ '**,**' ⟨*service_runtime_params*⟩
  | ⟨*empty*⟩

```
⟨parameter_runtime⟩ ::= ⟨logical_or_expression⟩

⟨constraint⟩ ::= 'constraint' ⟨service_signature⟩ ⟨compound_statement⟩

⟨architecture⟩ ::= 'architecture' '{' ⟨bindings⟩ '}'

⟨bindings⟩ ::= ⟨disconnections⟩
  |  ⟨connections⟩
  |  ⟨delegations⟩
  |  ⟨empty⟩

⟨connections⟩ ::= ⟨connection⟩ ';' ⟨bindings⟩
  |  ⟨empty⟩

⟨connection⟩ ::= 'connect' ⟨port_address⟩ 'to' ⟨port_address⟩

⟨disconnections⟩ ::= ⟨disconnection⟩ ';' ⟨bindings⟩

⟨disconnection⟩ ::= 'disconnect' ⟨port_address⟩ 'from' ⟨port_address⟩

⟨delegations⟩ ::= ⟨delegation⟩ ';' ⟨bindings⟩

⟨delegation⟩ ::= 'delegate' ⟨port_address⟩ 'to' ⟨port_address⟩

⟨port_address⟩ ::= ⟨identifier⟩ '@' ⟨component_identifier⟩
  |  ⟨identifier⟩

⟨component_identifier⟩ ::= ⟨collection_port_literal⟩
  |  '(' ⟨service_invocation⟩ ')'
  |  ⟨dereference_literal⟩
  |  ⟨identifier⟩

⟨collection_port_literal⟩ ::= ⟨identifier⟩ '[' ⟨expression⟩ ']'

⟨service_invocation⟩ ::= ⟨identifier⟩ '.' ⟨service_signature_call⟩
  |  ⟨identifier⟩ '[' ⟨index⟩ ']' '.' ⟨service_signature_call⟩

⟨index⟩ ::= ⟨identifier⟩
  |  ⟨constant⟩

⟨dereference_literal⟩ ::= '&' ⟨identifier⟩
```

Appendix **B**

New descriptors

```
descriptor Component {
    provides {
                default : *;
    }

    requires {
                args[] : *;
                owner : Component;
                descriptorPort : Component;
    }

    internally provides {
                super : * ofKind SuperPort;
                self : * ofKind SelfPort;
    }
    service getPorts() { ... }
    service getPortNamed(newName) { ... }
    service getDescriptor() { ... }
    service getOwner() { ... }
    service getIdentityHash() { ... }
}
```

Listing B.1: The Component descriptor

```
descriptor Descriptor extends Component {
  internally requires {
     name : String;
     parentName : String;
     ports[] : PortDescription;
     architectureDefinition[]: ConnectionDescription;
     services[] : Service;
  }

  service getName() { ... }

  service setName(newName) { ... }

  service getParentName() { ... }

  service setParentName(newParentName) { ... }

  service getDescribedPortAt(index) { ... }

  service getDescribedConnAt(index) { ... }

  service getService(selector, arity) { ... }

  service new() { ... }

  service newNamed(name, superDesc) { ... }

  service addService(serviceComponent) { ... }

  service removeService(selector, arity) { ... }

  service addPortDescription(pd) { ... }

  service removePortDescription(name) { ... }

  service addConnDescription(cd) { ... }

  service removeConnDescription(cd) { ... }
}
```

Listing B.2: The Descriptor descriptor

```
descriptor Port extends Component {
  requires {
        connectedPorts[] : Port;
        delegatedPorts[] : Port;
  }

  internally requires {
        name : String;
        interfaceDescription : Interface;
        isCollection : Bool;
  }

  service getName() { ... }
  service getInterface() { ... }

  service invoke(serviceName) { ... }
  service isConnected() { ... }
  service isDelegated() { ... }

  service connectTo(port) { ... }
  service disconnectPort() { ... }

  service isCollectionPort() { ... }
}
```

Listing B.3: The Port descriptor

```
descriptor CollectionPort extends Port {
  service invoke(serviceName, index) { ... }
  service disconnectPort(index) { ... }
}
```

Listing B.4: The CollectionPort descriptor

```
descriptor Service extends Component {
  internally requires {
  serviceSign : ServiceSignature;
        tempsN[] : String;
        tempsV[] : Component;
  code : String;
  }

  service getName() { ... }

  service setName(selectorName) { ... }

  service addParam(param) { ... }

  service getParamAt(index) { ... }

  service setCode(newCode) { ... }

  service getCode(newCode) { ... }

  service execute() { ... }
}
```

Listing B.5: The Service descriptor

```
descriptor ConnectionDescription extends Component {
  internally requires {
   sourceType : String;
   sourceComponent : String;
   sourceComponentIndex : UInt;
   sourceComponentInvocation : ServiceInvocation;
   sourcePort : String;

   destinationType : String;
   destinationComponent : String;
   destinationComponentIndex : UInt;
   destinationComponentInvocation : ServiceInvocation;
   destinationPort : String;

   bindType : String;
  }

  service setSourceType(type) { ... }
  service getSourceType() { ... }
  service setSourceComponent(scd) { ... }
  service getSourceComponent() { ... }
  service setSourceComponentIndex(index) { ... }
  service getSourceComponentIndex() { ... }
  service setSourceComponentInvocation(inv) { ... }
  service getSourceComponentInvocation() { ... }

  service setSourcePort(port) { ... }
  service getSourcePort() { ... }

  service setDestinationType(type) { ... }
  service getDestinationType() { ... }
  service setDestinationComponent(sdc) { ... }
  service getDestinationComponent() { ... }
  service setDestinationComponentIndex(index) { ... }
  service getDestinationComponentIndex() { ... }
  service setDestinationComponentInvocation(inv) { ... }
  service getDestinationComponentInvocation() { ... }

  service setDestinationPort(port) { ... }
  service getDestinationPort() { ... }

  service setBindType(type) { ... }
  service getBindType() { ... }
}
```

Listing B.6: The ConnectionDescription descriptor

```
descriptor Interface extends Component {
  externally requires {
    component : Component;
    services[] : Service;
  }
  internally requires {
    type : String;
    signatures[] : ServiceSignature;
    componentName : String;
  }
  service getType() { ... }
  service setType(type) { ... }
  service getSignaturesCount() { ... }
  service getSignatureAt(index) { ... }
  service addSignature(signature) { ... }
  service getConnectedComponentName() { ... }
  service setConnectedComponentName(name) { ... }
  service getConnectedComponent() { ... }
  service setConnectedComponent(component) { ... }
  service getServiceAt(index) { ... }
  service addService(serviceComponent) { ... }
}
```

Listing B.7: The Interface descriptor

```
descriptor PortDescription extends Component {
  internally requires {
    name : String;
    role : String;
    visibility : String;
    interfaceDefinition : Interface;
    kind : String;
    isCollectionPort : Bool;
  }

  service setName(name) { ... }
  service getName() { ... }
  service setRole(role) { ... }
  service getRole() { ... }
  service setKind(kind) { ... }
  service getKind() { ... }
  service setInterface(intf) { ... }
  service getInterface() { ... }
  service setVisibility(vis) { ... }
  service getVisibility() { ... }
  service setIsCollection(bool) { ... }
  service isCollection() { ... }
  service setType(type) { ... }
  service setComponentName(name) { ... }
}
```

Listing B.8: The PortDescription descriptor

```
descriptor ServiceInvocation extends Component {
  internally requires {
    receiver : String;
    selector : String;
    params[] : Component;
  }
  service setReceiver(receiver) { ... }
  service getReceiver() { ... }
  service setSelector(selector) { ... }
  service getSelector() { ... }
  service getParamsCount() { ... }
  service getParamAt(index) { ... }
  service addParam(param) { ... }
}
```

Listing B.9: The ServiceInvocation descriptor

```
descriptor ServiceSignature extends Component {
  internally requires {
    selector : String;
    paramNames[] : String;
  }
  service setSelector(name) { ... }
  service getSelector() { ... }
  service getParamsCount() { ... }
  service getParamAt(index) { ... }
  service setParam(param) { ... }
}
```

Listing B.10: The ServiceSignature descriptor

```
descriptor System extends Component {
  internally requires {
    name : String;
  }
  service println(string) { ... }
  service print(string) { ... }
  service readString() { ... }
  service readLine() { ... }
  service readInt() { ... }
  service getRand(seed) { ... }
}
```

Listing B.11: The Service descriptor

Appendix C

Acronyms

OOP Object-Oriented Programming

COP Component-Oriented Programming

CBSE Component-Based Software Engineering

ADL Architecture Description Language

DSL Domain Specific Language

MDE Model Driven Engineering

CORBA Common Object Request Broker Architecture

RPC Remote Procedure Call

IDL Interface Description Language

STL Standard Template Library

VM Virtual Machine

Appendix D

Contents of enclosed CD

```
sources....the directory with the archive with source files and md5 hash
 └─ dp_licek_tomas_2016.tar...............the archive with source files
     ├─ readme.txt.................the file with CD contents description
     ├─ compoVm........................the directory with C++ project
     │   ├─ coreModules..........the directory with core Compo modules
     │   ├─ include.....................the directory with C++ headers
     │   ├─ resources....the directory with Flex, Bison and Doxygen files
     │   ├─ script........................the directory with Perl scripts
     │   ├─ src..........................the directory with C++ sources
     │   └─ test.............................the directory with test files
     └─ thesis-text.......the directory with diploma thesis text sources
         ├─ make.pl........................the file with build Perl script
         ├─ resources..........the directory with pictures and other files
         └─ src..........................the directory with LaTeXsources
 └─ md5.txt..................the directory with the file with md5 hash
text..................the directory with the thesis text and assignment
 ├─ DP_Licek_Tomas_2016.pdf................the file with the thesis text
 └─ assignment.pdf.......................the file with the assignment
```