POZNAN UNIVERSITY OF TECHNOLOGY

Przemysław Adamek, Patryk Gliszczyński, Dariusz Lasecki

Integrated Development Environment for Automatic Verification, Synthesis and Improvement of Software

Bachelor's Thesis

Supervisor: Krzysztof Krawiec, Ph.D., D.Sc.
Advisor: Iwo Błądek, M.Sc.

Poznań, 2017

Abstract

This thesis focuses on the task of automatic verification, synthesis, and improvement of computer software. We give an overview of the field of Search Based Software Engineering and describe selected approaches, methods and libraries. As a technical part of this project, we develop a dedicated plugin for the Eclipse IDE which enables software engineers to easily use described tools in practice. The plugin relies on our original architecture which fosters extendability by providing generic mechanisms for integration with the wide range of external libraries. As a proof of concept, we implement support for Leon and Swim libraries. To validate the usability of our solution, we use it to perform a series of experiments that mimic common software development scenarios. Based on the experience gathered throughout this project, we share our own perspective on the future of the Search Based Software Engineering and suggest potential extensions for our plugin.

Acknowledgments

We would like to express our deep gratitude to our supervisor Krzysztof Krawiec and advisor Iwo Błądek for their continuous support and motivation. We sincerely thank for their participation and input, which certainly improved the quality of this thesis.

Contents

Chapter 1

Introduction

> The lurking suspicion that something could be simplified is the world's richest source of rewarding challenges.
>
> *Edsger W. Dijkstra*

1.1 Motivation

Software development is undoubtedly one of the broadest fields in modern computer science. It is the process of designing, prototyping, developing, reusing, modificating and maintaining widely understood source code. The individual needs for delivering new and maintaining old software are growing expeditiously every year. Hence the demand for engineers capable of performing this task increases respectively.

In the contemporary world, the process of automation is quickly spreading over almost every area concerning our everyday life. We already experience the progress of automation in fields that not so far ago required full human supervision. The cutting-edge technological examples may be found in domains related to navigation (e.g. self-driving cars), medicine (medical diagnosis), security (video surveillance), production (industrial robots), and commerce (automated retail). It should not thus come as a surprise that the advancement in automation will soon permeate more deeply in software development, letting the "machine" to take most of the responsibilities from the future developers. For example, engineers may be only required to define an abstract, high-level representation of the given problem, leaving the specifics of the implementation to an automated system. Nowadays, the automation in software development can be found mainly in the process of building software artifacts and their testing. However, there are many researchers working in this area, finding not only conceptual but also practical, new and innovative mechanisms for automation in the production of software. Throughout this thesis, we try to present the *state-of-the-art* approaches available and currently studied in the field of automatic verification, synthesis and improvement of software, as well as introduce our own contribution to this vast topic.

Currently, scientists working in this field are developing their own individual libraries to perform the operations mentioned above. For instance, there exists a range of software packages enabling synthesis of programs from examples or from specifications. These libraries usually differ significantly when it comes to the execution, collection of the results or presentation to the terminal user. Thus, from the perspective of a conventional developer, there is no standard way to communicate smoothly with such tools. This obligates

the developer to acquire appropriate *know-how* for each individual library in order to use it, as well as increases the learning curve and discourages developers from testing new techniques. In response to those challenges, we decided to create a unified software framework enabling both researchers and developers to use those functionalities intuitively from the higher level of the integrated development environment.

1.2 Scope and objectives

The main objective of this thesis is to implement a comprehensive *plugin*, dedicated for the *Eclipse IDE*, that will allow its users to effectively utilize the libraries implementing the methods of automatic program synthesis, verification and improvement in one unified tool. The plugin should be characterised by ease of use, even by a regular developer whose knowledge about automatic software development is not highly advanced. The more specific list of goals we would like to attain in this project includes:

- Design and implementation of generic interfaces for interoperation between the IDE and external libraries
- Clear presentation of the current processing status and results to the end user
- *Unix-like* operating systems support
- Seamless process execution and termination
- Demonstration of framework's application to exemplary synthesis and verification tasks
- If possible, elaboration of universal standards/plugins for connecting IDE with functionalities considered in this thesis and alike.

1.3 Organization

This thesis is divided into nine main chapters. Chapter 2 describes theoretical foundations of program synthesis, program verification, and software improvement, as well as explains what *genetic programming* is and how it can be successfully used for automation of software development. Chapter 3 demonstrates the technologies that were used in this project. It explains what *Scala* is and why is it so useful for search-based software engineering techniques. In this chapter one can also find a basic information about *Eclipse* - why we decided to expand the capabilities of this IDE and how the mechanism of modularity works in it. Chapter 4 focuses on libraries used for program synthesis tasks, which were used to test plugin capabilities. It presents a thorough explanation of what *Leon*, *Fuel*, and *Swim* are. In Chapter 5 we present the architecture of our project, including the brief explanation of main modules. The more comprehensive information regarding the implementation of each module can be found in Chapter 6. In Chapter 7 we present the experiments designed, implemented and performed for the purpose of this project. The thesis ends with Chapter 8 in which we sum up information and thoughts gathered while

working on this project, as well as describe the further steps which can be made in order to improve the presented application.

1.4 Contribution of the authors

This thesis was created by the team consisting of Przemysław Adamek, Patryk Gliszczyński and Dariusz Lasecki. Each member of the team contributed to essential technical developments achieved in this project, as well as to describing them and related theoretical aspects in this document.

Przemysław Adamek was responsible for back-end project development, in particular Abstract Syntax Tree, implementation of wrappers, project deployment and testing. Another important contribution was conducting experiments for the Search Based Software Engineering approximate methods and their comparison with exact methods.

Patryk Gliszczyński was responsible for the implementation of the generic system architecture, as well as the graphical user interface and integration with the Eclipse environment. As for theoretical aspects, he was involved in characterizing the overall system architecture including its implementation, and describing approximate methods of the Search Based Software Engineering.

Dariusz Lasecki was responsible for back-end project development, such as implementation of wrappers, parsers and communication with external libraries. Apart from implementation, he contributed to the theoretical foundations of the exact methods of the Search Based Software Engineering and related experiments, as well as conceptual aspects of the system architecture.

Chapter 2

Theoretical foundations

2.1 Program verification

Program verification is the process of using mathematical, formal methods to prove whether given code is correct and behaves as expected [1]. Both correctness and desired behaviour may be supplied in the form of specification, contract or exemplary test cases, depending on the programming paradigm and programming language.

Software can suffer from bugs of a very different nature. By classifying them into categories, their treatment can be easier and more systematic. Basic classes of software bugs are [2]:

- Runtime bugs - occur during the execution of a program; examples include division by 0, buffer overflows, null pointers etc.
- Functional correctness bugs - contracts, specification, relationship between input and output
- Concurrency bugs - happen when several entities try to access same resources; examples include race conditions, deadlocks etc.

Main approaches in code verification are [1]:

- *inductive assertions*

  In approaches based on inductive assertions, control points are established throughout the code with certain assertions that show the relationship between input variables and program variables that should be satisfied at each control point. If all possible paths of execution are captured in these assertions, then the problem of correctness is actually the matter of proving certain formulas (verification conditions).
- *functional semantics*

  The functional semantics method is based on transforming the flow of the program into a mathematical function that maps input to output. This transformation can be in certain cases done automatically. Once the formula is defined, it can be proved using mathematical methods, such as mathematical induction.
- *explicit semantics*

In explicit semantics, computer programs are considered as logical objects, by which we mean that they are explicitly expressed in the logic of choice. Commonly, programs are represented as *tree-like* structures, because this makes the task of encoding them in the logic easier. Semantics of a program is defined in the form of axioms. They may, for example, state that a program with a variable assignment instruction is equivalent to the same program with a value substituted for every occurrence of this variable. Inference rules of the logic, together with stated axioms, give ground for verification of a specified program.

2.2 Program synthesis

Program synthesis can be considered as an instance of a search problem [3]. Given a set of all possible programs and conditions for correctness, the task is to find at least one program that fulfills the conditions. The set of all programs is usually infinite and not given explicitly in the form of objects that would form a search space, therefore it is more convenient to operate with the concept of a programming language which we treat as a tool that is capable of expressing any program. A programming language introduces its own constraints on how programs can be formulated and introduces basic building blocks such as variables, data structures etc. In such a case, the process of synthesis can start from a simple piece of code (which possibly does not fulfil the given requirements) and gradually change it according to the rules imposed by the programming language, so that program quality (e.g., the number of passed tests) improves. Depending on the goal, context and resources, synthesis may lead to a program which satisfies the whole or the part of a given specification.

The problem of correctness in program synthesis can be approached in multiple ways. Depending on the point of view and a particular context, one can come up with many definitions for correctness. The very general one is that a program works according to user's intent. Indeed, every program that exists has been brought to life to satisfy some kind of a need that originated in the mind of the user or its creator. However, this definition is ill-formed in the sense that it is difficult to grasp formally; one can argue that the human intent is prone to being imperfect and erroneous. It also brings some relativity to the table in depending on what we understand by user intent. Ideally, we would like to specify the one, well-established user and its intent that captures all of the possible test cases and thus guarantees that the program is correct. It naturally leads us to the concept of the oracle which is capable of telling whether the program satisfies all the necessary conditions. Unfortunately, the oracle is a concept that operates in a black-box manner and its practical implementation brings us back to the problem of how to formally define the correctness of a program.

To eliminate this vicious cycle, the correctness conditions are practically defined as either passing a set of given tests or conformation to a given formal specification. It is worth noting that the former is highly dependent on the quality and coverage of test cases provided. Usually, it is impossible to enumerate all possible test cases and thus the synthesized program is guaranteed to be correct only within the universe of provided examples which reflect user's hidden intent. The latter however can often be proved

formally for any allowed input.

Program synthesis comes with many benefits. First of all, it delegates the effort of creating software (code) from a programmer to an automated tool. A programmer is only responsible for providing some kind of specification and possibly providing the synthesizer with some hints, whereas the tedious work of translating the specification into a piece of code is waived. From the business point of view it saves resources like time, money, number of people. It also means that if a universal standard for providing specification was proposed, then developers would not be required to learn numerous programming languages and technologies to implement software, because a specification expressed in such a universal way could be translated into any existing programming language. Most importantly, the synthesized programs could be (depending on the nature of specification) provably correct, i.e. produced with certificates of correctness, which means that they certainly work as intended. Apart from correctness, various automated optimization techniques could accompany the process of synthesis to improve the non-functional requirements, like memory or processor usage, power consumption, length etc.; for detailed description of improvement of programs see Section 2.3 below.

2.3 Program improvement

Program improvement is the act of improving some of program's characteristics, and can for example be used for:

- improving time or memory efficiency (program optimization) [4]
- reducing energy usage (for example improving battery life of battery powered devices) [5]
- fixing bugs in a program [6]
- reducing required network bandwidth or network usage [7] [8]
- code refactoring - improving code readability or reducing code complexity [9]
- adding more automatic tests to improve code coverage [10]

When improving some of the non functional characteristics of a program it is important to preserve the functional correctness of the program.

Program improvement can be automatic, and there were many notable achievements in automatic program improvement in recent years, to name few:

- MiniSAT - using generic improvement and code transplants resulted in creating a new evolved version of MiniSAT that was on average 17% faster than the original 2009 MiniSAT solver. [11]
- repair functionality of Leon [12]
- genetically improved version of BarraCUDA [13], which is a project that uses CUDA graphics card to map DNA reads to the human genome [14]

2.4 Genetic programming

In *artificial intelligence* (*AI*), an *evolutionary computation* (*EC*) technique called *genetic programming* (*GP*) is used to automatically resolve problems without a necessity to define the problem-specific form or structure of the desired solution beforehand [15]. The idea of evolutionary algorithms reaches back to 1950s when *"several computer scientists independently studied evolutionary systems with the idea that evolution could be used as an optimisation tool for engineering problems"* [16]. The original goal was not to design algorithms to solve specific problems but rather to *"formally study the phenomenon of evolution as it occurs in nature and to develop ways in which the mechanism of natural adaptation might be imported into computer systems"* [16]. In genetic programming the population of computer programs is being evolved. A GP algorithm transforms populations of programs, generation by generation, into new successive populations of programs that are expected to perform better according to a given quality criterion (*fitness*). It leads to finding a suboptimal solution for any given problem by the metaheuristic approach.

GP focuses mainly on evolving computer programs represented as tree structures [17] (though alternative representations have been researched there too). A program tree consists of internal nodes which represent the operator functions, as well as terminal nodes that define the operands. This representation is relatively similar to the *Abstract Syntax Tree* (*AST*). Figure 2.1 depicts an example of a program tree.

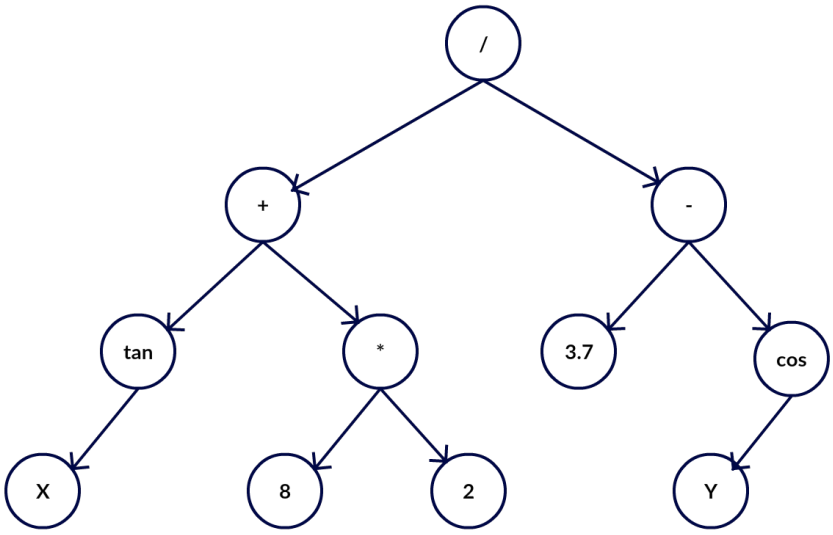

Figure 2.1: Example of a program tree.

The tree structure is a very convenient representation for manipulation (modifications) and evaluation. The strategy predominantly used in the field of modern GP research is based on generic *evolutionary algorithm* (*EA*), which consists of five major executional steps which are respectively adapted for the program evolution and evaluation purposes. The diagram in Figure 2.2 presents the whole process with the description followed below.

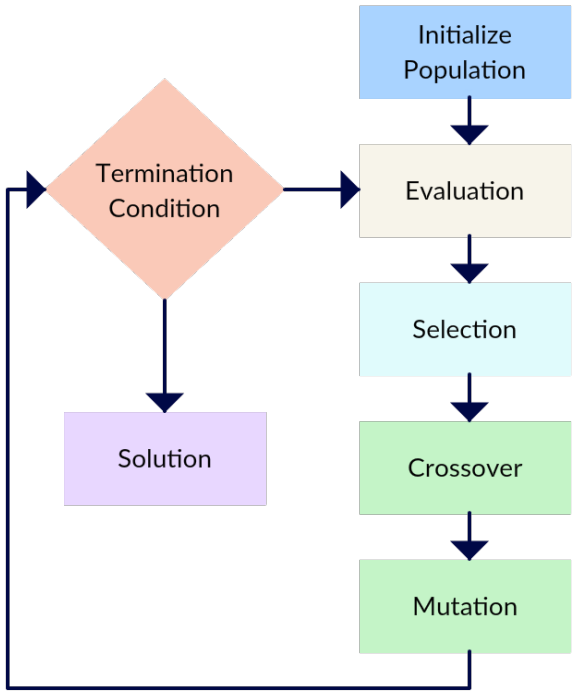

Figure 2.2: Steps in the generic evolutionary algorithm.

2.4.1 Population initialization

In the first place, an initial population of programs is (typically randomly) generated. There are many different methods of initialization, though the overall goal is to spawn the source population either entirely randomly or based on known properties of the desired solution. An example initialization technique involves recursive building of a program tree in a top-down manner, up to a given tree height limit.

2.4.2 Evaluation

In this step, each individual solution in the population undergoes evaluation by a fitness function. The notion of fitness in this context corresponds to loss function in the optimization problem. The fitness function in GP may focus on evaluating the solutions based on their accuracy, the number of tests that they are able to pass or looking for the programs that are consuming less resources, like CPU or memory.

2.4.3 Selection

As with most evolutionary algorithms, an important stage of workflow is selection, in which the well-performing (*fit*) individuals are chosen to become 'parents' for the next generation of candidate solutions. This process is typically randomized, so that on one hand the best-performing individuals are not guaranteed to be always selected, while on the other it is not entirely impossible for the worst performing ones to become parents. Typically, programs with higher fitness have increased probability of survival. This method may be compared to the mechanism of natural selection, wherein the features of more fit

organisms are favored to be preserved in further generations. The method of selecting *phenotypes* is a broadly developed area in evolutionary algorithms. There have been many different approaches proposed to this task, however the most commonly used one in GP is called *tournament selection*. In tournament selection, a predefined number of individuals is randomly selected from the population. From this group, only the best one (based on its fitness value) is being selected. In order to select more than one candidate solution, the tournament selection can be run multiple times. Usually in this step we will be trying to select more than one candidate solution because they may be necessary in the next step of the algorithm.

2.4.4 Search operators

In conventional GP the next population is constructed from individuals from the current population by combining results from two genetic operators: *crossover* and *mutation*.

Crossover (*recombination*) is the process of creating one child solution from at least two parent solutions. Typically, a child solution shares an original part of each of its parents. The inspiration for this operator comes directly from the biological process of reproduction where chromosomes (genes) of the parents are shared with their offspring. There are many different ways to perform the recombination, though the logic behind them is similar. The necessary part is to choose at least one *crossover point* which would indicate the parts within the parents and then mix the information from both parents, in order to form at least one representative child. This idea can be conveniently illustrated for individuals represented as *fixed-length vectors*, which we do in Figure 2.3.

Figure 2.3: Example of a crossover in genetic algorithm.

Another fundamental genetic operator - mutation, is used to maintain the chromosomes heterogeneity from one population of candidate solutions to the next one. It means that by the process of mutation we try to transform one individual into another one by changing its structure in a random, albeit non destructive manner. There were many different approaches proposed for this task. The most intuitive and simplest one is called *bit-flip* mutation, where the operator takes a chosen individual, picks a random (and typically short) range of its bits and inverts them accordingly to the given representation. The example for this method is illustrated in Figure 2.4. The extreme variant of this operator is *one-bit* mutation that inverts a single bit.

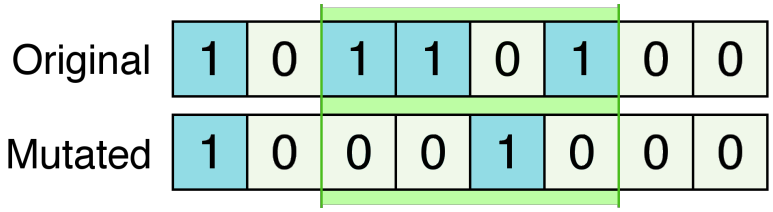

Figure 2.4: Example of a bit-flip mutation.

GP deviates substantially from other evolutionary algorithms in the implementation of the search operators like crossover and mutation, which is enforced by the unusual representation of candidate solutions in this genre of evolutionary computation [15]. As we mentioned earlier, the program in GP is most commonly represented as a tree structure; for this reason different methods for crossover and mutation need to be devised. The most popular one is called the *subtree-swapping* crossover. It takes two parental trees, divides them into many subtrees by defining at least one crossover point in the operator node and, analogically to the standard crossover, swaps the parental subtrees, creating the child program. Figure 2.5 depicts the subtree crossover example in which two parents mix the randomly selected subtrees together in order to form their offspring.

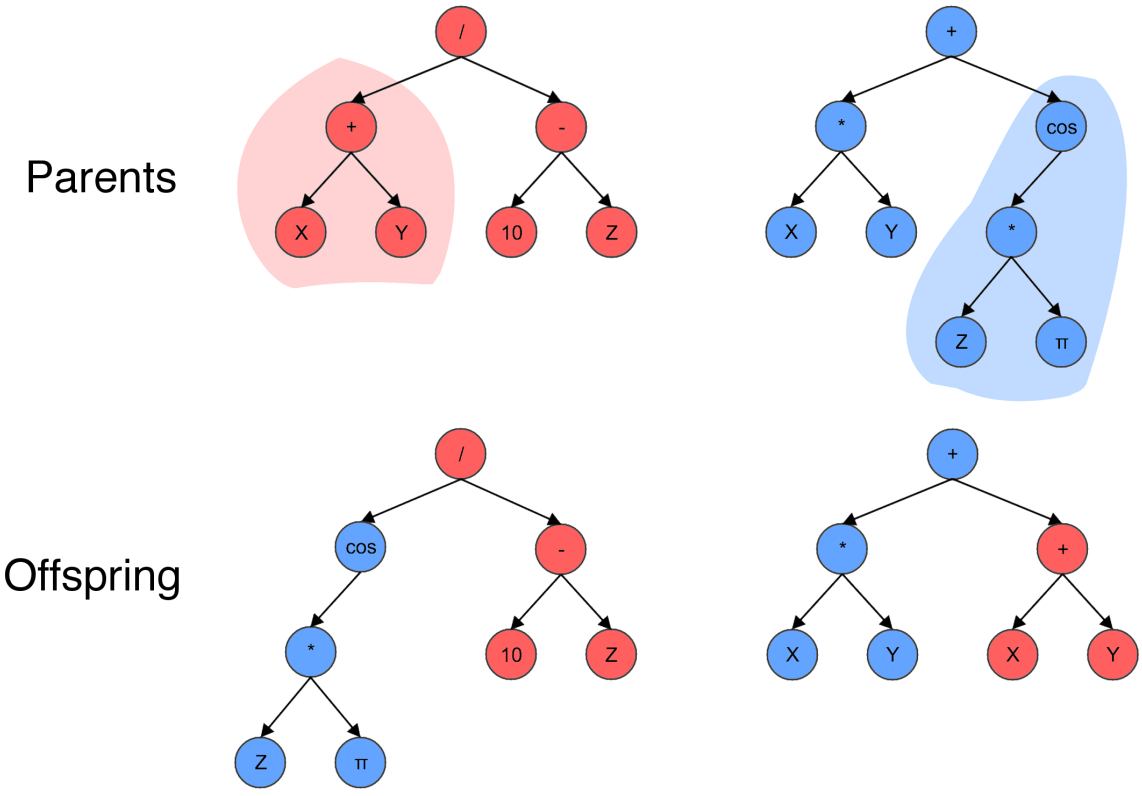

Figure 2.5: Example of a subtree-swapping crossover.

Mutation operator applies a similar logic to the crossover, using technique called *subtree* mutation, which replaces a subtree that was randomly selected with another randomly created subtree [18]. An exemplary outcome of this method is presented in Figure 2.6.

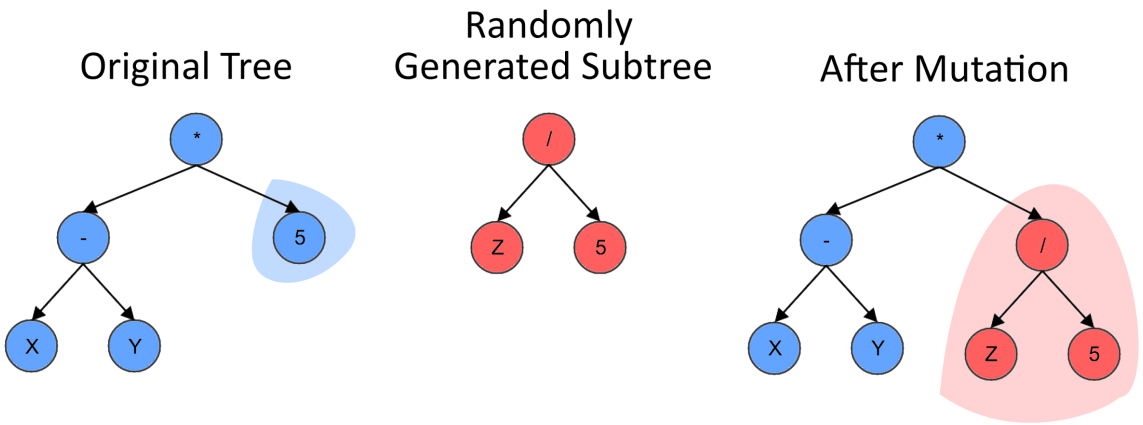

Figure 2.6: Example of a subtree mutation.

2.4.5 Termination

The final step is to check whether any of the termination criteria is met. If so, then the algorithm is stopped and the best solution is returned. Otherwise, the algorithm moves on to the next generation, starting again with the evaluation part. Termination criteria may differ from one use case to another. Generally, the simplest termination criteria includes checking whether the maximum number of generations elapsed, reaching maximum processing time or obtaining satisfying enough (fit enough) solution.

In our project, we used an external genetic programming library, which will be thoroughly described in Chapter 4.

Chapter 3

Technical background

3.1 Scala programming language

Scala is a programming language developed at the *École Polytechnique Fédérale de Lausanne* (*EPFL*) and published in 2004 [19]. Despite its relatively young age, it turned out to be extremely successful in both academic and industrial environments. Its strength lays in merging the *object-oriented* and the *functional* paradigms of programming.

The object-oriented design of Scala language is reflected in the fact that every value is treated as an object and every operation is a call of a method (strong resemblance to Java) [20]. As a result, every entity in the code can be treated in a standalone manner. Objects may contain data in the form of attributes and code in the form of methods. In this setting, computer programs can be treated as sets of objects interacting with each other.

Functional programming is a flavour of the *declarative programming* paradigm. Its basic characteristics are the lack of data mutability and state changing. Programs are then written with expressions and declarations rather than statements that are typical for imperative languages. These features mean that the result of a function is only dependent on its input and not any internal states that may have changed between function calls. This feature is also known as absence of the so-called 'side effects'.

Similarly to other popular languages like Java or C#, Scala can be classified as a statically typed language - types of variables must be clearly indicated (unless automatically derived by the compiler from the context) and communicated to a compiler. Thanks to this, many bugs such as types mismatch can be caught at a compile time. When it comes to the aforementioned Java and C#, it is worth mentioning that Scala possess quite similar syntax and thus coexists well in projects using those imperative languages. The motivation of creating the Scala language was, however, to eliminate the shortcomings of Java and C# which are often blamed for making the process of creating the component software troublesome.

The *'component software'* is a term that reflects the idea that software should be ultimately built from ready-made components such as libraries, web services, packages etc [21]. It would make the whole process of creating software faster and more efficient. In reality, quite often certain generic components are written over and over again by different teams for particular products. This not only hampers productivity but also makes software

more error-prone. Specialized, well-established libraries are certainly much more tested than their ad-hoc counterparts. Advocates of Scala claim that the technological *status quo* is a result of programming languages created in a way that does not foster the component approach. Thus, Scala, as its name indicates, is focused on providing scalable tools that can be applied at any level - both in small and large systems. Together with the emphasis on such mechanisms as abstraction, composition and decomposition and the celebrated blend of object-oriented and functional paradigms, Scala aspires to be the programming language which will ignite the component-based era in software engineering.

For the purpose of this thesis however, we should mention the features of Scala that are particularly useful for the search-based software engineering techniques. These are surely Scala preconditions which consist of *assert*, *assume*, *require* and *ensuring* functions. Generally, they aim to provide dynamic invariant checking, static code analysis and serve for documentation purposes. They accompany declarations of functions to enforce that certain conditions should or must be fulfilled. Their functionalities are operationally equivalent, albeit react differently to failures - by throwing different types of errors and messages:

- Assert defines a condition that has to be proved by the code analyzer in the compile time and throws an error in case of failure to do so.
- Assume defines an axiom for the static code analyzer that can be used to prove assertions and has to hold. An error is thrown if the assumption does not hold.
- Require and ensuring throws an exception whenever any method call violates the condition given. The former is used as a precondition and the latter as a postcondition.

In the context of search-based software engineering, these functions can be used for synthesis, verification and improvement of programs. In case of the verification, the existing body of a method can be checked against specified conditions. When it comes to the synthesis, conditions stated allow the synthesizer to come up with a code that meets indicated requirements. Similarly, during the process of improvement, the code undergoes changes that must guarantee conditions to still hold.

3.2 Eclipse environment

Our plugin is targeted towards users who primarily write code in Scala. We were looking for good a IDE with Scala support and two promising options were identified:

- IntelliJ IDEA with Scala plugin
- Eclipse with Scala plugin

Choosing between the two was a matter of personal preference, and we chose the latter option.

Eclipse is one of the most popular applications for software development on the market, being extensively used both by professional engineers and researchers. It has an impressive documentation and community. According to *'StackOverflow Developer Survey 2016'* [22] where tens of thousands of software developers were asked, it was the second most

commonly used IDE in 2016 (excluding advanced text editors, which were also included in the statistics). The current version of Eclipse (4.6 *Neon* released on 22 June 2016) already has over one million downloads [23]. Eclipse IDE supports multiple programming languages (e.g. Java, C, C++, Scala, PHP, JavaScript, Scala) and runs on a wide range of operating systems - e.g. *Windows*, *Linux*, *macOS*. Eclipse uses a weak copyleft license [24] known as *'Eclipse Public License'* that makes Eclipse both open source and free software.

Eclipse is actively being developed and releases so called *'simultaneous release'* each year [25]. The next version, 4.7 *Oxygen*, is expected to be released in June 2017.

3.3 Concept of plugins

Eclipse is a very modular piece of software, created primarily with the intent that every component can be freely modified or extended. The original Eclipse application consists of many predefined base components. A component in Eclipse is called a *plugin*. In Eclipse IDE, even the bottom-line functionalities are delivered as plugins. Naturally, the Eclipse platform allows developers to freely extend the Eclipse-based applications, like Eclipse IDE, with supplementary functionalities via the mechanism of plugins [26]. Because of that, it is possible to contribute to the existing components, e.g. by defining the additional toolbar or menu entries, preparing custom perspectives or specifying new features.

Eclipse applications are based on a runtime module called *Equinox* [27]. It is a subproject of the Eclipse project which allows developers to implement applications from the set of smaller modules, called *bundles*. Equinox is based on a modular system that implements a dynamic component model, called *Open Service Gateway Initiative* (*OSGi*) [28]. OSGi is a Java framework used in the process of development and deployment of software that mainly focuses on building applications from the set of already prespecified modules in order to achieve full separation for specific components. It enables the components to be dynamically added, removed, activated, deactivated and updated. Thanks to this mechanism, it is possible to extend the basic capabilities of Eclipse platform easily and effectively, even in runtime. The ideology of modularity is rooted deeply in the implementation of Eclipse, thus we were substantially motivated to construct our own plugin for this exact IDE.

The Eclipse foundation offers a digital store, called *Eclipse Marketplace*, in which the community-created plugins are stored. It delivers a straightforward integration with Eclipse IDE to browse and install new plugins directly from the IDE seamlessly. As of writing this thesis, it has almost 2000 different solutions (plugins) ready to be downloaded, with over 23 million installations been already made directly from Eclipse.

Chapter 4

External libraries

4.1 Leon

4.1.1 Introduction

Leon is a system for software verification, synthesis and repair developed by the *École Polytechnique Fédérale de Lausanne* (*EPFL*) [29]. It is a tool that helps developers in building verified software written in Scala programming language.

Leon provides both a command line and a web interface to interact with its tools. The web interface is available on the web page *https://leon.epfl.ch/*. It interacts with users via the web-based code editor and allows execution of its functions on a remote server. The command line version requires installing Leon on one's own machine accompanied by several dependencies such as external *SMT* (*satisfiability modulo theories*) solvers, Scala language distribution and *Java SE Development Kit*. Once installed, the verification, synthesis and repair tasks can be executed locally. This type of interaction with Leon has been adopted in our project in which the Eclipse plugin communicates with Leon via a command line.

4.1.2 Pure Scala

Leon operates on the subset of the official Scala language, called *Pure Scala* [30]. It is the subset defined by the creators of Leon. Although the assumption that programs are written in Pure Scala may be constraining, the subset offers a great variety of essential data types and structures so that the assumption is not so limiting for developers using Leon. The list of predefined types supported in Pure Scala includes:

- Boolean
- TupleX
- Int
- BigInt
- Real
- Set

- Functional Array
- Map
- Function

The features available in Pure Scala include:

- boolean conditions
- algebraic data types
  - abstract classes; limited to these without fields or constructor arguments
  - case classes; extension of an abstract class, includes fields
  - case objects; extension of an abstract class, without fields
- generics; classes and functions with generic types
- methods; defined in classes or defined as abstract methods in abstract classes, methods can be overridden
- specifications
  - preconditions; implemented with the 'require' instruction
  - postconditions; implemented with the 'ensuring' instruction
  - body assertions; implemented with 'assert' within the function's body
- expressions
  - pattern matching and custom pattern matching
  - values
  - inner functions; functions defined inside other functions

Since Pure Scala is as a matter of fact a subset of Scala, it is easily handled by widely available Scala compilers.

4.1.3 Verification

Verification of software is the major functionality delivered by Leon [31]. Actually, the whole system started as a verificator and developed other features as the platform matured. Verification relies on Scala preconditions and postconditions. As mentioned in the Section 3.1, these correspond to *'require'* and *'ensuring'* clauses. Assuming that there exists a Scala function to be verified, with syntactically correct body and proper precondition and postcondition, Leon is capable of verifying whether the existing implementation meets specified requirements. To prove the correctness of a function, Leon uses the combination of an internal algorithm and external SMT solvers. It is worth noting that this approach is exact in the sense that it can result in one of the following outcomes:

- postcondition is valid - any input to the function satisfies the conditions
- postcondition is invalid - there exists at least one input that violates the conditions and the concrete counterexample is presented

- postcondition is unknown - Leon did not manage to find any counterexample nor to prove the correctness (usually occurs as a results of an internal error or a timeout)

In other words, results produced by Leon verification are mathematically strict.

Apart from verifying conditions provided by users, Leon includes also several code safety checks techniques . These techniques are, among others, checking array accesses to ensure that they are in valid bounds and checking full coverage of pattern matching cases. The algorithms used by Leon are independent of those used by the Scala compiler and are thus capable of detecting different kind of issues that may be undetected by the Scala compiler.

4.1.4 Synthesis

Another important feature of Leon is program synthesis [32]. In contrary to verification, it assumes that there is no complete implementation of the function considered. There are two options for specifying a synthesis task – using the 'choose' construct or a hole depicted by '???' instruction. The 'choose' construct allows the user to explicitly state the specification of the program to be synthesized and encapsulates the specific condition for the output to satisfy. In this case, there is no function implementation at all and the code in synthesized solely based on the condition specified. The '???' option serves for the purpose of synthesizing a small fragment of a larger existing implementation. The conditions that must be met are specified via postcondition and precondition (require and ensuring instructions).

Formally, a synthesis task is defined in terms of:

- The input variables which are within a choose statement
- The output variables that should be synthesized
- A path-condition which constrains input variables
- A specification that describes the relationship between input and output variables

Such stated problem is supposed to be converted by Leon into an executable program and a precondition that ensures that the generated program is correct.

Since synthesis is a very challenging problem, Leon uses special decomposing rules which are supposed to simplify the whole process. The basic idea is to split the whole task into smaller sub-tasks. These sub-tasks are evaluated independently of each other but there always exists a precise instruction on how to merge them at the end into a valid solution to an original, complex problem. Decomposing rules implemented in Leon include:

- equality split

  For two input variables x and y of proper types, two test cases are generated:

  $$\text{x == y and x != y.}$$

- inequality split

  For two input variables x and y of numerical types, three test cases are generated:

  $$\text{x} < \text{y, x == y, x} > \text{y.}$$

- ADT split

  For a variable of an algebraic data type, a rule is decomposed into sub-rules each concerning the subtype of the algebraic data type.

- int induction

  For an Int or BigInt variable, a base case sub-rule is generated as well as one sub-rule for each inductive case (helpful for recursive methods).

- one point

  If an output variable is assigned a value and that output variable occurs in later parts of the specification, that value is substituted for the output variable.

- assert

  Looks for constraints on input variables included within the specification (i.e. choose structure for Leon synthesis), moves them out of the specification and creates a precondition corresponding to that constraint. That precondition is assumed to be equivalent to the removed constraint.

- case split

  It is responsible for decomposing top-level disjunctions into separate cases.

- equivalent input

  Searches for redundancy in input variables and resolves them. It covers equivalences of variables at the top level as well as those inferred from the Abstract Data Tree (ADT).

- unused input

  Detects variables included in the choose structure that are not part of any constraints in the specification. Such variables are considered useless and are eliminated.

- unconstrained output

  Detects output variables that are not constrained and synthesizes them with a trivial value of a given type, for example 0 for integers.

4.1.5 SMT solvers

Leon relies heavily on external solvers to verify and synthesize programs. Currently supported SMT solvers include *CVC4* and *Z3*. CVC4 is an open-source SMT solver developed in cooperation with the *New York University* and *University of Iowa* [33]. Z3 is an SMT solver developed by *Microsoft Research* [34]. SMT solvers are tools for solving decision problems for logical formulas assuming certain theories which are formulated in classical first-order logic with equality [35]. Typical examples of such theories, supported by the aforementioned solvers, are theory of real numbers, theory of integers or theory of more complex objects like arrays, vector, tuples, strings etc. Conditions provided for Leon are usually in this form, therefore such solvers are naturally capable of proving whether required conditions hold for certain parameters. Delegating these tasks to external solvers, that are well established and powerful tools, is a practice that ensures top-notch performance and high level of reliability.

4.2 Fuel

FUEL is a compendious open-source framework created for the task of implementing metaheuristic algorithms, principally evolutionary algorithms. It is natively written in Scala, preserving the functional programming paradigm of immutability for most of its classes. As it was stated by the author, *"it can be particularly useful for using metaheuristics in innovative ways, e.g., for hybridizing them with other algorithms or devising 'homebrew' algorithms"* [36].

FUEL's main role is to help developers building metaheuristic algorithms from the existing components available in the library. It provides a user-friendly way of parametrizing algorithms, facilitating the deployment, and simplifying the process of collecting results.

4.3 Swim

According to Swim library's webpage:

> SWIM is a compact library written in Scala that implements the basic functionality of Genetic Programming. To realize the functionality of evolutionary algorithm, SWIM relies on the FUEL library which implements the evolutionary computation workflow in Scala. SWIM provides the GP-specific components for that workflow, i.e., mainly data structures for representing programs, random generation of initial candidate programs, search operators and evaluation. [37]

Since Swim implements typed GP, it can be used both for solving problems where instructions return values of different types and for solving simpler single-typed problems. The default evaluation method (fitness function) is the number of failed tests (which Swim tries to minimize) [37]. Like Fuel, Swim is licensed under the MIT License.

To test Swim in our project we used BooleanFromCSV example provided by the authors of the Swim library. Although we did not come up with the BooleanFromCSV ourself, we created many different inputs for the problem - see the Experiments section. BooleanFromCSV is an example of synthesis from examples using GP. The application takes a path to a CSV file with examples as an input. The CSV file should contain one example per line. All examples in the file use the same number of input variables. Each example consists of n boolean values (for n input variables) and one desired (expected) output, also a boolean value. All values should be separated by a delimiter. Using GP, Swim tries to find the best function defined for n ($n \in N$) boolean variables that satisfies most of the examples provided (ideally all). Allowed operations in the generated function are:

```
!& (nand), | (or), & (and), !| (nor).
```

Result - the generated function is presented in the prefix notation with variables being numbered with consecutive natural numbers - 0, 1, 2, etc. Example output:

```
|(&(|(!&(0 0) 1) !&(!|(0 0) |(1 1))) &(&(!&(0 1) &(1 1)) 0))
```

Chapter 5

System architecture

From a broader perspective, there are four main modules that should be distinguished in our system's architecture: *User Interface*, *Task Executor & View*, *Library Mediator* and *Wrappers*. Each of them plays a distinctive and crucial role in the overall system. The general concept of a system architecture is presented on the diagram in Figure 5.1.

5.1 User Interface

The User Interface module focuses on the contribution to the Eclipse IDE. In this part we extended the capabilities of a standard Eclipse IDE by defining our own *perspective*, adding new entries to the *main menu* and editor's *context menu*, as well as creating our own visual component for the *workspace view*. This module is responsible for the interaction with the end user, enabling to perform certain actions from the IDE level. As it was specified on the diagram in Figure 5.1, there is a group of *handlers* responsible for the execution of new specific tasks. These handlers are activated whenever a user explicitly starts a new task of automatic software development. Handlers are also used to collect the required input data that is further forwarded to the Task Executors.

5.2 Task Executor & View

Task Executor & View module is a bridge connecting the User Interface and Library Mediator. It consists of two main submodules - *Task Executor* and *Task View*.

- Task Executor is constructed when a user refers to a handler. It represents the specific *order* dispatched by a user. By an order we understand a specific task of automated software development, like program synthesis, verification or improvement, and the data linked to it, e.g., list of class methods, program code or a pointer to specific file on disk. Task executor should recognize what kind of task user wants to start and check whether all of the required data for this specific task were provided by the handler. If so, then it refers to the Library Mediator to execute the concrete order.
- Task Executor creates at least one Task View for the specific order. Task View always refers to only one class method in a context of a single order. Because a

Task Executor may receive more than one class method to process in a single order, it may happen that it will create more than one Task View. Task Executor keeps the reference to all of its Task Views and if any data comes back from the Library Mediator, it will inform the adequate Task View about it. Task View is responsible for the communication with the User Interface. It is initiated by a Task Executor and holds the reference to its creator. Task View's role is to inform the User Interface module about a current processing status, as well as respond to the user requests, e.g., stopping tasks. In that case, it will refer to the Task Executor in order to send a specific signal to the Library Mediator.

5.3 Library Mediator

Library Mediator module is an intermediary between Task Executors and Library Wrappers. It keeps references to the wrappers and responds to the calls coming from the Task Executors. There are two main parts of the Library Mediator.

- *Library Watcher* checks whether any new wrappers have been installed, or if the existing ones have been modified or deleted. Should any of this happen, it performs a specific action in order to instantiate or finalize the wrapper, and informs the UI about it.
- *Library Manager* stores the initialized library wrappers. When it receives a new order from the Task Executor, it reads the received order metadata and delegates the task to the proper wrapper. It is also responsible for waiting for the new data returned by the wrappers, and for forwarding received data to the appropriate Task Executor.

5.4 Wrappers

Wrappers module is the lowest level part of our system's architecture. The main role of a wrapper is to create a communication standard between the Library Mediator and an external library. By external library we understand a SBSE library like Leon or Swim. Wrappers are defined by the external users that are willing to connect their own libraries to our plugin. Wrappers are based on an abstract class that already implements the communication with the Library Mediator. Users part is to specify what actions the library can run, and how to execute those actions after receiving the specific signal from the Library Mediator. The more detailed information about the wrappers can be found in next chapter.

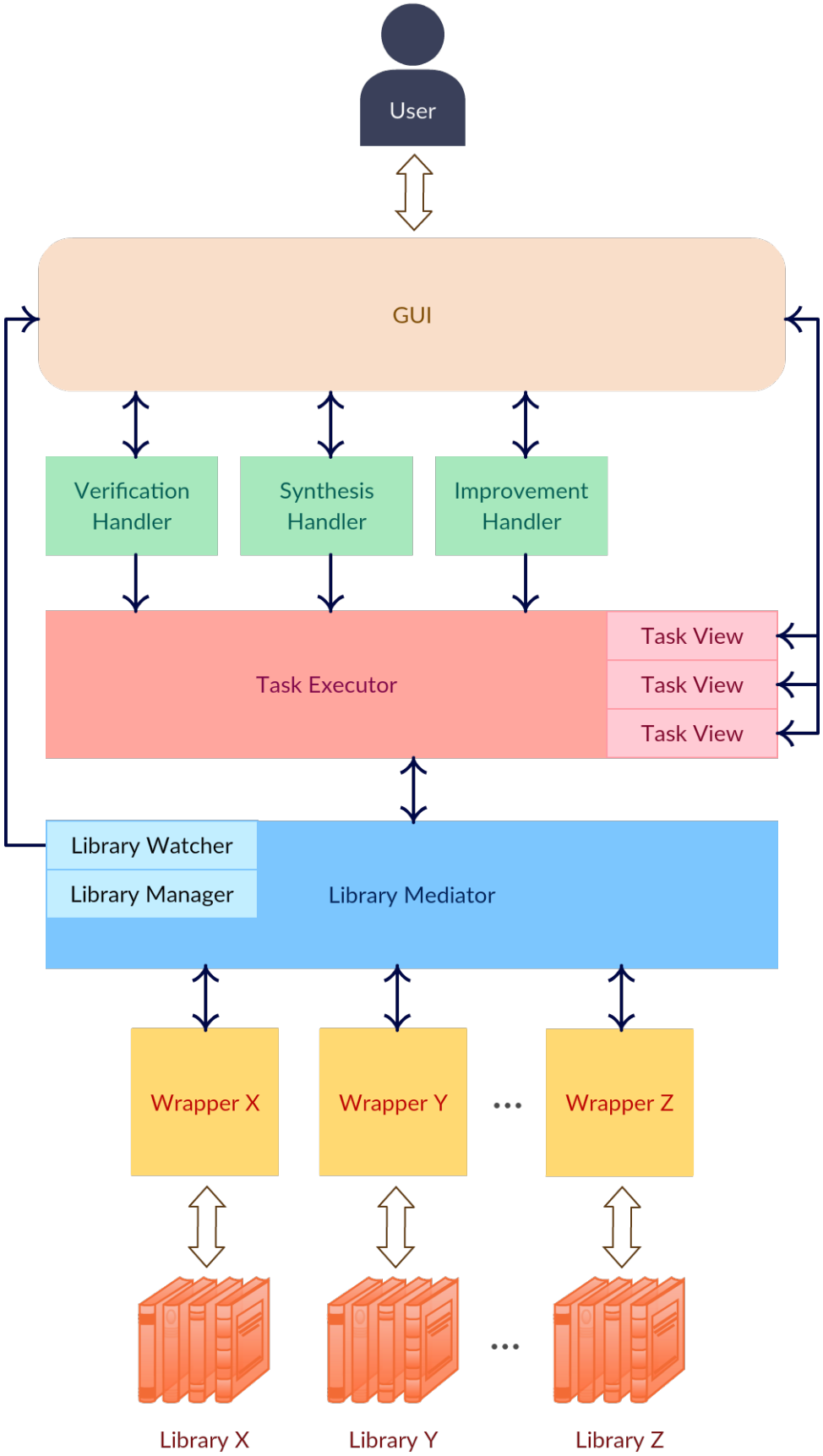

Figure 5.1: Our system's architecture diagram.

Chapter 6

System implementation

To implement our plugin we used a dedicated Eclipse IDE called *Eclipse IDE for Eclipse Committers*. It is a package suited for developers planning to contribute to Eclipse applications. In this chapter, we describe in more detail the implementation of the constituents of our system.

6.1 User interface

Graphical User Interface (*GUI*) is delivered to the end user through the mechanism of plugins. As mentioned in Chapter 4, Eclipse offers a wide spectrum of functionalities that can be implemented, extended or modified. In our plugin we extended the standard Eclipse IDE with new menu entries, defined an individual perspective, implemented a custom workspace view and designed a special dialog. Individual contributions are described in the sections below.

6.1.1 Perspective

To clearly separate the development process from the software evolution part we decided to create a dedicated perspective. According to the Eclipse documentation:

> A perspective defines the initial set and layout of views in the Workbench window. Within the window, each perspective shares the same set of editors. Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources. [38]

It allows us to specify in which context and places do we want to make our UI contributions visible to the user. Thanks to that, the user does not see our menu entries or custom views in other perspectives that he is typically using. Otherwise, it could be distracting and overwhelming for the developer. As it is depicted in Figure 6.1, all the user has to do in order to see our GUI contributions is switching to our perspective called Program Synthesis. The empty perspective view is presented in Figure 6.2.

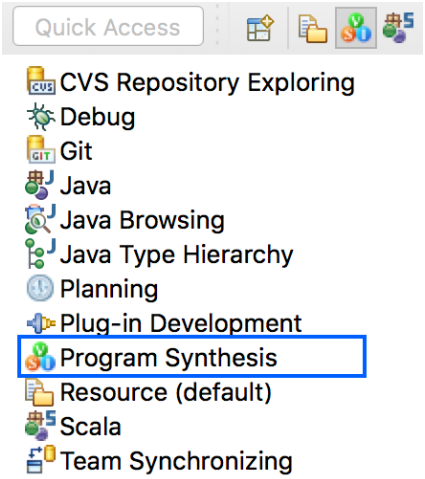

Figure 6.1: "Program synthesis" perspective visible among others in the Eclipse IDE.

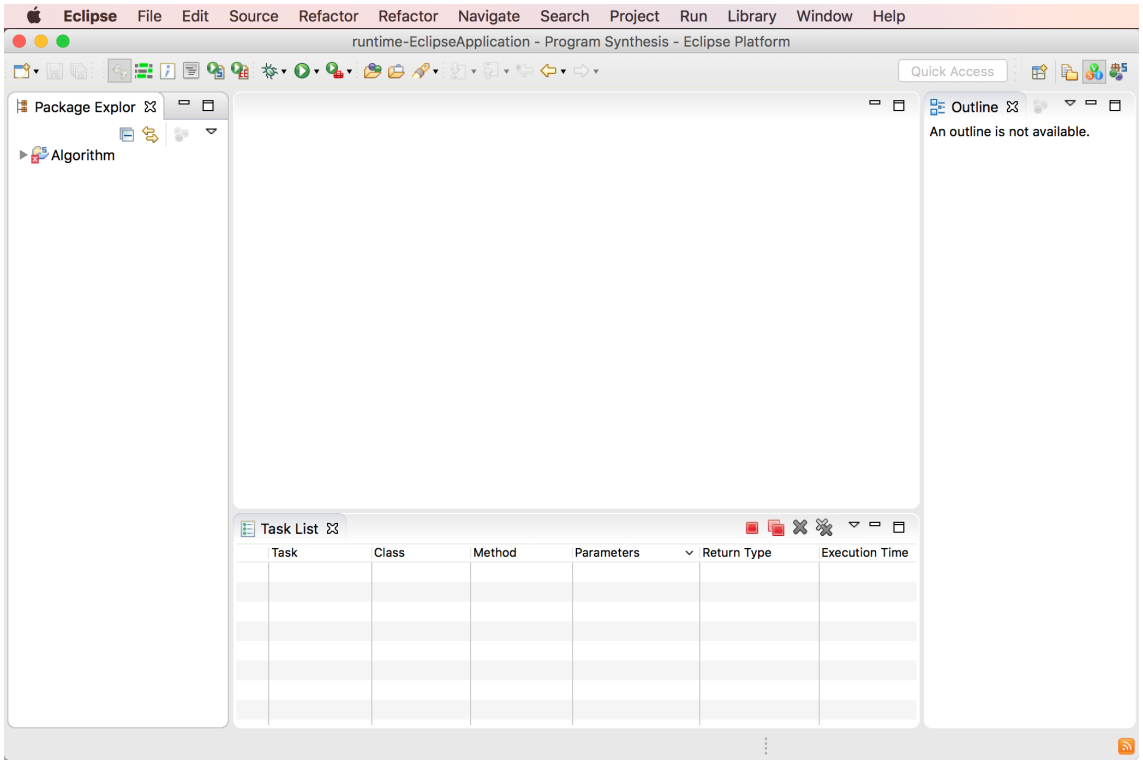

Figure 6.2: The appearance of our perspective in the Eclipse IDE once it is activated.

6.1.2 Menu entries

To give user a possibility to select libraries and actions provided in our plugin, we created two new entries in the main (Figure 6.3) and context (Figure 6.4) menu. The user can either click on an entry in the main menu or right click on the editor to select the library or action. Additionally, we decided to allow the user to pick an action only when he has a focus on the editor. Therefore, if the editor is not opened or it has no focus, the action menu entry will not be visible. It is motivated by the necessity to run actions in the

context of a class. The only scenario when it's not necessary is with the program synthesis from the examples. In that case user is not obliged to open any class, but has to point to a specific file containing the expected examples.

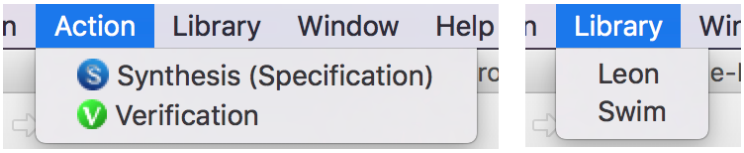

Figure 6.3: Menu entries in the toolbar available to the user through our perspective in the Eclipse IDE.

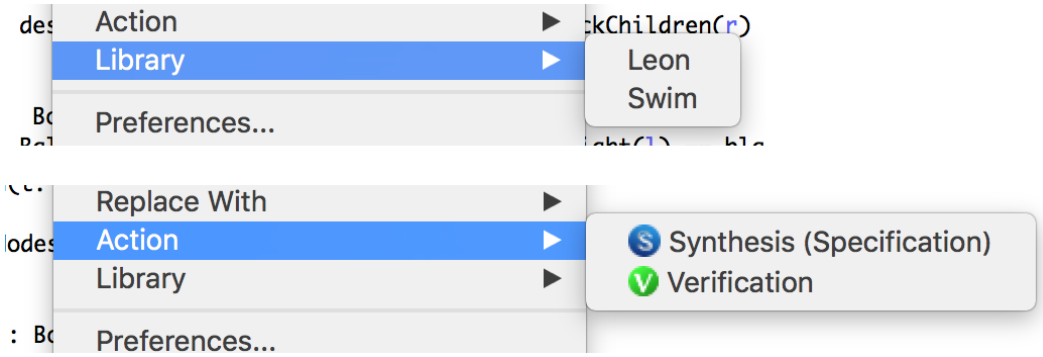

Figure 6.4: Menu entries in the context menu available to the user through our perspective in the Eclipse IDE.

6.1.3 Custom list view

Naturally, the actions executed by the user should be in some way visually presented to the user. Thus we decided to implement our own custom view called a Task List. Task List presents user the current processing status of all recent orders, as well as the possibility to perform certain actions:

- kill the currently running tasks,
- remove tasks from the list,
- open the details dialog by double clicking on the selected position on the list.

Task List is using a *TableViewer* class to render the list. Each row on the list corresponds to one Task View, and one Task View corresponds to one class method. As it was described in Chapter 5, each order supported by a Task Executor may consist of many Task Views. It means that for a single order there might be many entries on the list. Therefore we decided to indicate each order by assigning the group number to it. A single entry on the list is a graphical representation of a Task View, which stores the current processing status (*running*, *success*, *fail*), type of the task (*verification*, *synthesis*), class and method name, list of parameters, return value type, and the execution timestamp. Those fields are depicted on the list in separate columns.

It is worth mentioning that the data visualised on the task list is connected with a Task View by a mechanism of *data binding* [39]. Thanks to this mechanism, whenever an associated field changes its value, the User Interface module will be notified about it and the displayed value on the list will be respectively updated. In addition, we added a possibility to sort individual columns by clicking on their headers. It gives user a chance to arrange the list accordingly to his personal preferences. This component is presented in Figure 6.5.

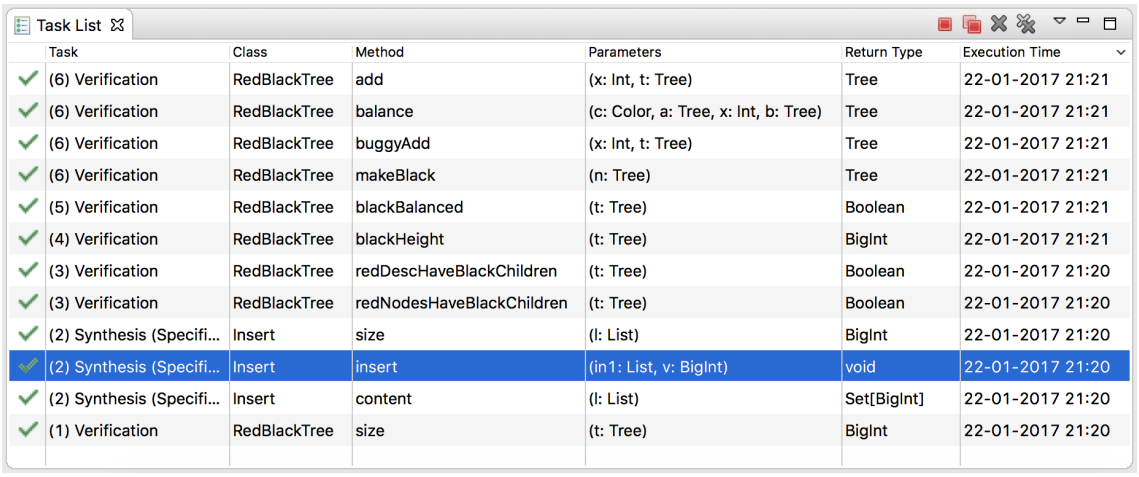

Figure 6.5: Custom task list available to the user through our perspective in the Eclipse IDE.

6.1.4 Task details dialog

To provide the user with more detailed information about an individual task, we implemented a special dialog. This dialog is displayed to the user after double clicking on an entry on the task list. Besides the information presented on the list, a dialog includes a *raw console log* and a *results log* containing only the most important information. To implement the dialog we used a popular graphical widget toolkit for the Java platform called *SWT* [40]. Similarly to the list entry, dialog is also binded with a Task View object. Thanks to that, whenever new results come back from the wrapper, the information in the dialog is refreshed. In order to give to the wrappers' creators more control over how their results are presented to the user (e.g. bold, underline, etc.). Task details dialog interprets some of the ANSI escape codes. ANSI escape codes are a standard in the industry, they are fairly easy to use and many tools are already using them (e.g. Leon). The example dialog for the verification task is presented in Figure 6.6.

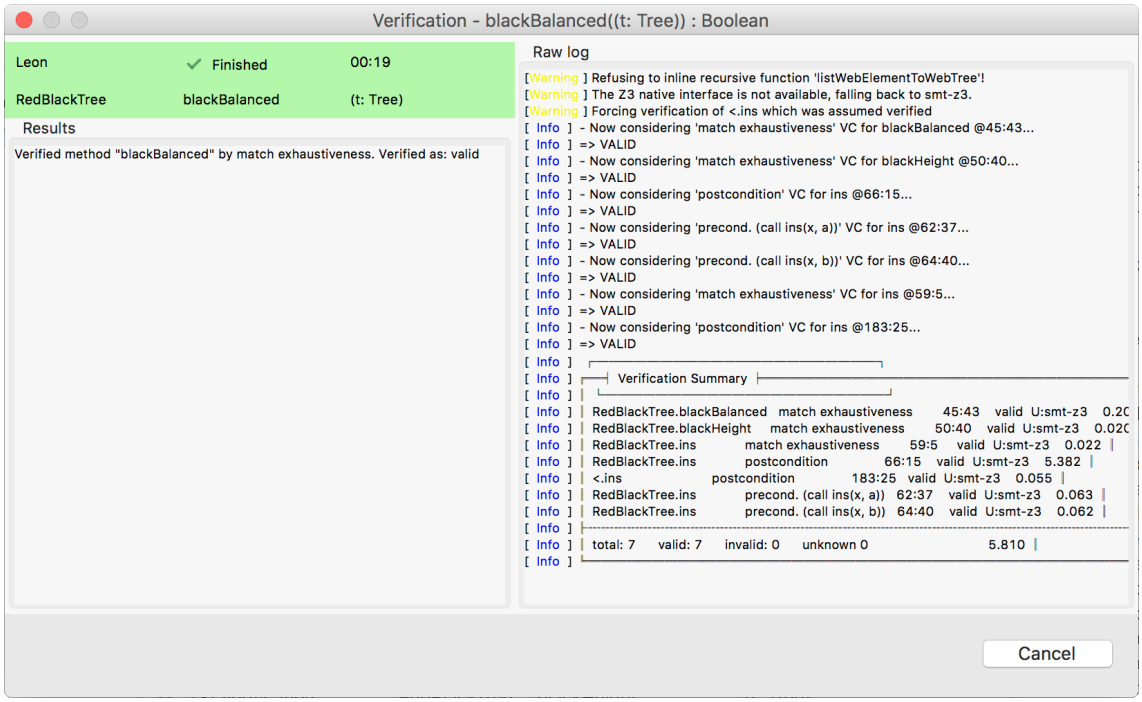

Figure 6.6: Example of a dialog of a verification task from our plugin.

6.1.5 Handlers

As described in Chapter 5, Task Handlers are special classes used to communicate with Task Executors in order to begin the specific task. In our application we distinguish handlers responsible for starting program synthesis from the specification, program synthesis from the examples, and code verification. Before referring to the Task Executor, each handler needs to collect the input data. We distinguish two different cases in which we collect task-specific types of data.

In the case of synthesis from examples, a file dialog is opened that asks user to select files from his drive, that he wishes to use for synthesis.

In the case of synthesis from specification and verification, task handler:

1. Gets the path of currently opened file in active editor.
2. Gets AST of only selected code fragment (in case of no selection, AST of the whole file is taken).
3. Starts a new task and passes the file path and the AST to it.

6.1.5.1 Abstract Syntax Tree

Abstract Syntax Tree (*AST*, or *Syntax Tree*) is a simplified representation of a source code that most often is expressed by a tree-like structure. *"Each node of the tree denotes a construct occurring in the source code"* [41]. The difference between AST and *Concrete Syntax Tree* (*CST*, or *Parse Tree*) is that an abstract tree has the freedom to not show every detail of the source code [42].

Need for the AST

AST is useful when it's necessary to programmatically access or modify information in source code in a consistent and convenient way. This was the case in our plugin, when we needed to extract the information about methods and classes selected by the user from a source code. The general workflow of the plugin is that a user selects a code fragment in a suitable file with intent for some program (library) to perform some operations on this code. Handlers take into account the selected code, extract from it the information about classes and methods, and pass that information to the Library Mediator. Apart from that, we also wanted to present to the user information about running/finished tasks in a nice way. We decided that it would be useful to present information about selected class, method, method's arguments and method's return type for each task (see Figure 6.5).

Getting the AST

We need to get the AST from a Scala source file selected by the user. Through our research we found a couple of ways to achieve this:

- Scala reflection utilities
- Implementing custom Scala compiler plugin
- Using an external library (most of them being some part of parsers integrated into IDEs - eg. NetBeans, Eclipse, IntelliJ).

We chose the last option and used part of Scala IDE Plugin for Eclipse that is responsible for creating the AST [1]. To be exact, we used the part that is responsible for creating AST which is later used by the Outline view. Although the Outline view does not offer all of the capabilities that you are used to that ASTs often offer (what you see in the Outline view is what you get, literally - see Figure 6.7), it certainly can be classified as AST and was enough for our needs.

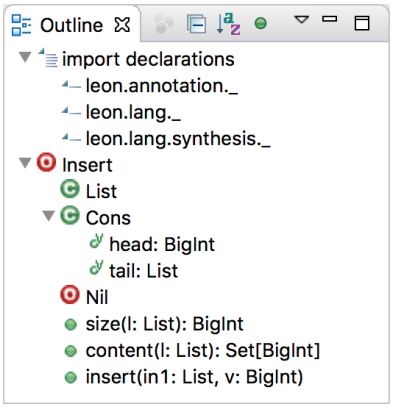

Figure 6.7: Outline view of the Eclipse IDE - the source of AST information.

Relying on Scala IDE plugin seemed like a good idea, because the user is obliged to have the Scala plugin installed anyway to be able work with Scala source files in Eclipse.

[1] that part of the plugin also happens to be tremendously underdocumented

The only drawback is that the Outline, and therefore our AST, works only on opened projects that are imported to the workspace. It will not work for files unassociated with projects, but since that use case is very rare we did not consider this as an issue. There exists a working workaround for this problem (that we learned by analyzing ScalaIDE plugin's unit tests) to create a temporary Scala project, copy the source file to it, and get AST out of the file that is now inside a Scala project and delete the project. We have not implemented that workaround, instead we wrote a fallback function that gets only names of selected functions (no information about classes, functions' parameters and arguments) using regexp matching.

One of the packages that we needed to use (*org.scalaide.ui.internal.editor.outline*) is not on the Export-Package list in the MANIFEST.MF file of the *org.scala-ide.sdt.core* bundle, which meant that it was not available for use outside of the bundle. We fixed that by writing a script that adds that required for our plugin package name to the Export-Package list in the MANIFEST.MF file that is inside the *org.scala-ide.sdt.core* bundle. We kindly ask users to run that script during installation of our plugin (see Appendix A.1).

Up to this point, our plugin was written primarily in Java. Since the chosen library for retrieving AST is written in Scala, we had to add Scala aspect to the project. We decided to keep all of the AST logic in Scala to keep code readable and short. We created a wrapper class in Scala that can be later easily used in Java code. That wrapper class retrieves AST from a file and filters only the information that we are interested in.

6.2 Task Executor & View

6.2.1 Task Executor

Task executor's role is to take care of a particular order initiated by a user. It implements four main methods:

- *execute* - initializes the prescribed order. It takes as a parameter an *enum* value representing the task's type, and a map containing input data. Firstly, it checks whether the library supports the multiple methods execution in a single order. This means that a library is able to synthesize, verify or improve many class methods in a single library call. For instance, Leon provides this kind of support which results in notably better overall performance when we compare running many methods in a single order, to the situation where we split an order into many suborders. However, if this feature is not implemented, then a single order has to be respectively divided, hence for each individual method an additional Task Executor must be created. This method also creates at least one Task View. In a situation when a library supports the execution of multiple methods in a single order, for each method an individual Task View will be created. Finally, it forwards the order to the Library Mediator module.
- *kill* - sends a specific signal to the Library Mediator module, informing it to stop the appropriate order. Because Task Executor may hold many Task Views in a single order it is obliged to inform all of its child Task View objects about its finalization.

- *setResults* - is called by a Library Mediator module whenever new processing results are generated. Its role is to propagate received data among the existing Task Views.
- *getData* - gathers the data from all of its Task Views.

6.2.2 Task View

Task View object takes care of the communication with the User Interface. It represents one class method for a single order and shares its processing status with the UI. As it was mentioned earlier, a Task View object is sharing its fields with the UI by the mechanism of data binding. For this task we implemented a *PropertyChangeSupport*, which notifies other objects supporting this mechanism (in this case the Task List and Details Dialog) about the changes happening in a Task View. Task view implements three methods:

- *kill* - is called after user clicks a special button on the Task List. It refers to the appropriate method in Task Executor and finalizes the Task View instance, which results in its disappearance on the Task List.
- *setResults* - is called by a Task Executor. Its role is to update the proper fields in a Task View with the new received information.
- *getData* - collects all the information from the Task View fields. It is usually called by a Task Executor object.

6.3 Library Mediator

The Library Mediator acts like a global agent in the communication between Task Executors and Wrappers. It is used to load, instantiate, and operate on the Library Wrappers provided by external users. Together with wrappers it creates a universal transmission mechanism allowing to directly reference to the external library. It consists of three main parts.

6.3.1 Library Watcher

Library Watcher is an object responsible to monitor a certain directory (or directories), looking for the library wrappers. We decided to store our program dependencies under one directory */opt/psynth*, keeping wrappers in a separate subdirectories, e.g., */opt/psynth/leon*, */opt/psynth/swim*. The monitoring part is based on a class in Java called *WatchService*, which implements the low-level communication with an operating system, in order to be notified whenever inside the observed directory file is being created, modified, or deleted. Thanks to this mechanism, we are not obliged to periodically check whether some changes occurred in a specific directory, but rather we register a *Watchable* object which will be notified by the operating system if any action has happened. It is not only helpful for the developer, but also improves an overall performance since there is no need to constantly recheck the directory state. We defined one main listener for the /opt/psynth directory, which checks if any folders have been added, modified, or deleted.

- For every new folder created in the /opt/psynth directory, an additional listener is set up, which in this case listens for the *Java Archive* (*JAR*) files. If inside of this subdirectory a JAR file was found, it creates a new Library Instance with an URL pointing to a JAR file as a constructor's parameter.
- In a case when either a JAR file or the whole directory was deleted, Library Instance is finalized and unnecessary listeners are removed.
- When a directory or a JAR file is modified, both listener and Library Instance are removed, and then created again.

Thanks to that mechanism, we are able to add, delete, and reload our JAR files during runtime. Naturally, at a first run of the application we check if any folders already exist in the /opt/psynth directory, and whether they contain JAR files.

6.3.2 Library Instance

The library object represents a single instance of the external library provided by a user. It is directly connected to a wrapper, and plays an important role in the communication with it. As it was mentioned in the previous subsection, it is initiated by a Library Watcher. In its constructor Library Instance receives an URL pointing to a specific JAR file representing the wrapper. With that URL, Library Instance is trying to add specific class to the classpath, and also instantiate a wrapper by a mechanism of reflection. The wrapper instance is then casted to an abstract class called *IWrapper*, which will be described in the next section.

In case of failure, e.g., the file pointed by an URL does not exist, or it will not be castable to the IWrapper class, the appropriate exception will be thrown and the library will not be added to the Library Watcher's list. Otherwise, the library will be initiated and from this moment it will be possible to call certain methods supported by a wrapper.

6.3.3 Library Manager

The role of the Library Manager is to redirect requests coming from Task Executors to the proper Library Instances. It stores a list of all Library Instances, and implements the method execute. When a Task Executor decides to begin an order, it refers to this method. Library Manager transfers the order to the proper library.

6.4 Wrappers

Wrapper is an adapter connecting the external library with a Library Instance. It is primarily implemented by an independent user, and is based on an abstract class called IWrapper. For the sake of this project we implemented two separate wrappers supporting Leon and Swim. However, any other external libraries can be connected by this mechanism.

6.4.1 IWrapper

IWrapper is an abstract class which already comes with the implementation of methods used in a communication with the Library Mediator (*start*, *stop*, *getData*). It demands from a user to define certain methods independently for the specific needs of his library. In its constructor as a parameter it takes a config map describing the functionalities of the library. A user is obliged to implement two abstract methods:

- *run* - takes as a parameter a map containing data describing a specific order. Its role is to respond properly to this order by executing the appropriate method in user's library and sending back the results to the proper Task Executor. The results can be sent either when returning from the function call, or can be updated repeatedly with the use of method update, which notifies the proper Task Executor about the altered data.
- *finish* - stops the process of the particular order, accordingly to the user implementation.

IWrapper also defines four enum types, which are helpful in communication:

- *TaskType* [SYNTHESIS_EXAMPLES, SYNTHESIS_SPECIFICATION, VERIFICATION]
  Specifies what kind of task user wants to start.
- *Status* [RUNNING, SUCCESS, ERROR]
  Defines the current order processing status.
- *Inport* [CODE, CLASS, METHOD, EXAMPLES]
  The possible input data for the specific order
- *Outport* [STATUS, CODE, COUNTEREXAMPLE, CONSOLE_LOG, RESULT_LOG]
  The possible output data that can be returned from one order

6.4.2 Parsers

Every external library that can be incorporated into our plugin reports their output to user. The structure of this output varies from one library to the other and it is impossible to define any universal parser that would handle generated reports and select only parts that are the most relevant ones. It is often the case that libraries generate many technical information and essential parts like synthesized code or found counterexamples have to be found among them. It is not comfortable in everyday activities. Thus, we equipped our plugin with the capability of defining custom parsers for every library. The process of writing a parser is optional and can be done during the development of a particular wrapper. Provided interfaces give possibilities of reporting both parsed and raw output to the GUI. This solution allows the user to see both clear output produced by their own parser and raw output in case they need more information.

Our implementation provides wrappers for Leon and Swim libraries. To present full capabilities of our plugin, we equipped them with suitable parsers as well. They are based on simple pattern matching techniques and exploit structures recognized in libraries' outputs. In the Swim library output we bold the synthesized function and remove less

relevant information like the name of the computer, Java version, process id etc. In the Leon library verification output we extract function names, final status of the verification and potentially found counterexamples. We remove redundant information and descriptive statuses reported during the computation. Results of our parsers versus the raw output can be observed by comparing 'Results' and 'Raw log' columns in 6.8 and 6.9.

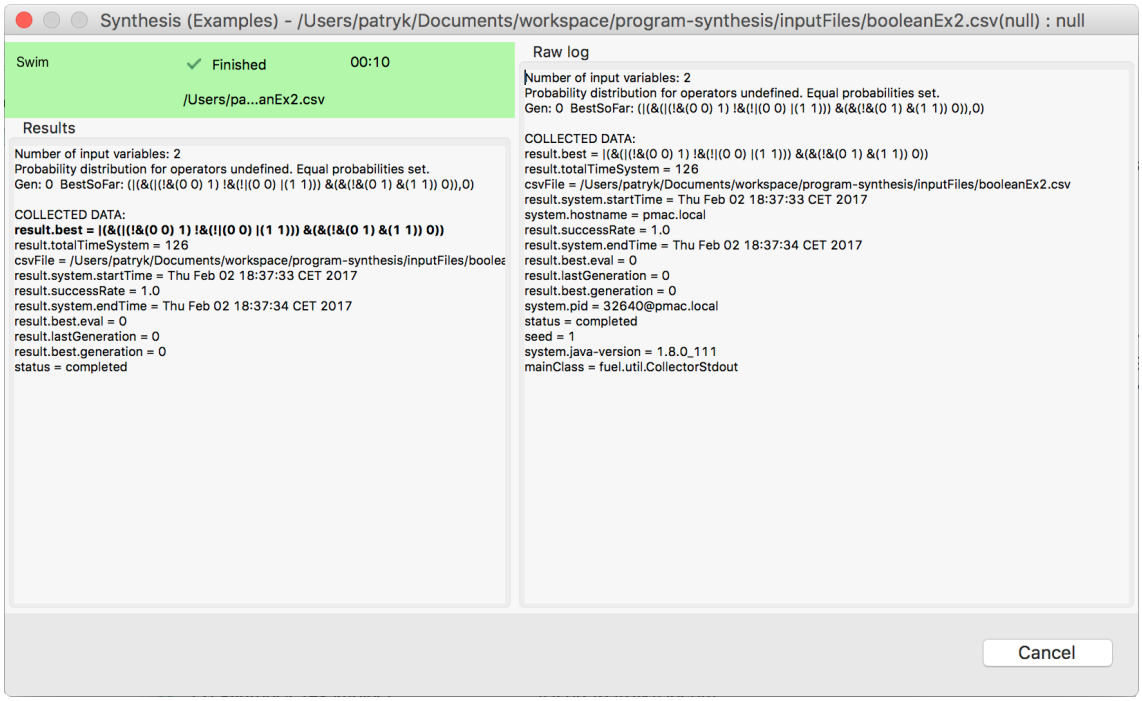

Figure 6.8: Output of a Swim synthesis from examples task - parsed output on the left and raw output on the right.

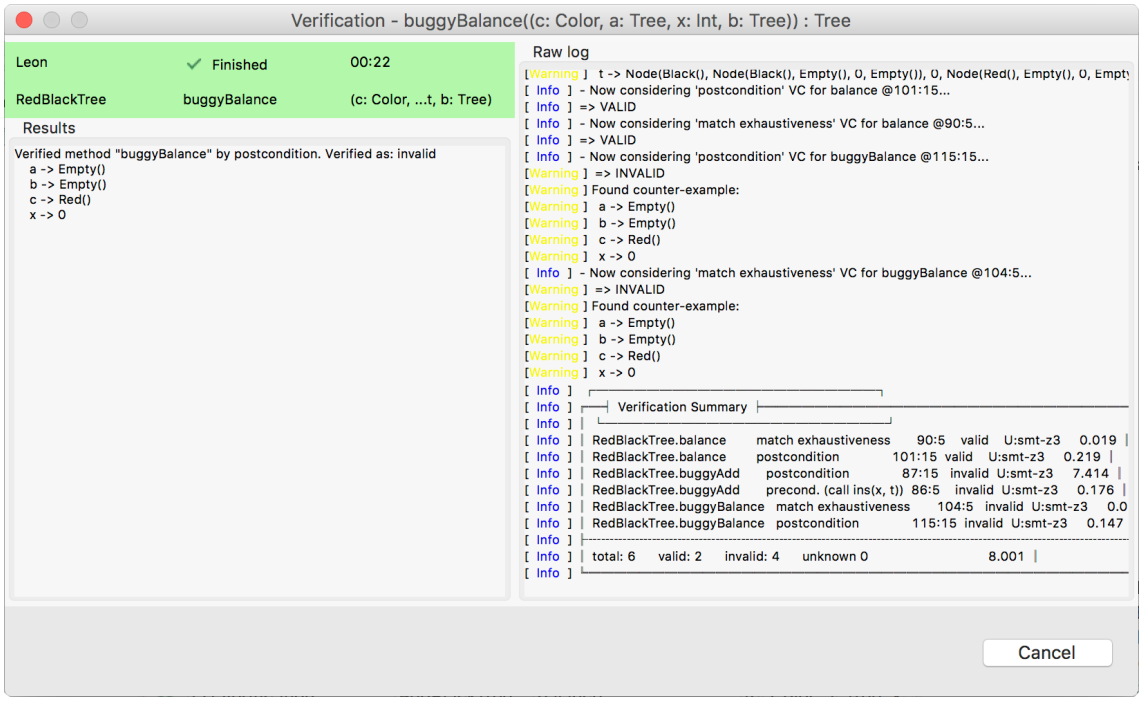

Figure 6.9: Output of a Leon verification task (single method) - parsed output on the left and raw output on the right.

6.4.3 Stopping tasks

The user should be able to send a request to stop a selected task at any time they likes. This might be particularly helpful because tasks may run longer than the user expects, or the user might have simply changed their mind and is no longer interested in the result of a task. We also need to be prepared to stop all tasks when the user decides to close the Eclipse IDE, otherwise the user might be left with high CPU usage caused by tasks that were left running.

Whoever writes a wrapper also needs to implement task's *finish* method. This method is called when a task needs to be stopped. In both Swim wrapper and Leon wrapper we implemented the *finish* method in the same way. Some required preparations need to be done before starting tasks. Both Swim and Leon are started using Java. There is an option to provide java with additional arguments (even if they are not used in any way by the program). We decided to use a 'name' (*java -Dname=name_given_by_us*) argument. Although this argument is not used by the program, it helps us to find the started process later. As long as we can guarantee that this 'name' parameter is going to be unique in the whole system, we will be able to find the process we started anytime we like. This is done by listing all of the running processes and filtering them by the name command line argument that we provided when starting the process.

To stop a program we first find its process id. *Process ID* (*PID*) can be obtained from the job's name - this functionality is provided by the IWrapper abstract class. Once the PID is obtained, we send (using a kill command) a signal (either SIGTERM or SIGINT) to the process. It is the authors of the Leon and Swim programs responsibility to handle the signal and exit their program gracefully. If sending SIGTERM and SIGINT does not cause the program to quit, we are forced to send SIGKILL as the last resort. *"This signal causes immediate program termination. It cannot be handled or ignored, and is therefore always fatal."* [43]

6.4.4 Leon Wrapper

We created our Leon wrapper with support for verification and synthesis from specification. We put that piece of information in Leon wrapper's configuration settings, so that Library Manager knows what functionalities to present to the end user.

Whenever verification or synthesis is requested, wrapper is given some data that explains user's intent:

- path to the file selected by the user
- list of methods selected from a file (and optionally the class that each method belongs to). If there are multiple methods with the same name in the file and only a subset of them has been selected, it is useful to have information about the parent class for each method to differentiate between them. Without information about the class that a method belongs to, Leon runs verification/synthesis for every method with a matching function name.

We start a Leon job by using a Leon bash script (which runs Leon with all the appropriate settings) that is created during Leon installation. The user needs to ensure that the leon

command is systemwide accessible (that requirement with instructions on how to achieve that is listed in the Appendix A.1). Without making leon script systemwide accessible, we would not know the location of the the leon script. Because of that we would have to ask the user to put that information in a configuration file. The first option seemed less burdensome and more user-friendly to us.

We pass to the Leon bash script a path to the selected file and information about selected methods (if available we also pass information about the classes that each method belongs to). When using synthesis we also add the –synthesis flag to tell Leon to run synthesis instead of the verification which is the default option.

Leon wrapper saves leon jobs' logs to /tmp/psynth/leon directory (we redirect stdout and stderr to a file). After the job is done and results are presented to the user, logs are removed. *"On most systems, /tmp directory is cleared out at boot or at shutdown by the local system."* [44] Thanks to the choice of /tmp directory to store logs in, should anything go wrong (failure to remove logs by our wrapper), logs will be probably removed anyway during the next reboot.

While Leon task is executing and incremental progress is made by Leon (and saved to the log file), we immediately try to present the progress to the user. This has been done by registering (in another thread) a WatchService on /tmp/psynth/leon directory. Whenever a file changes in this directory and it matches the name of our log file, new data is read from the file and pushed to the wrapper's *update* method.

Once the job is finished, the updates thread (the "live log thread") that was monitoring Leon progress (registered WatchService) is going to be stopped and parsed results will be sent up the plugin's chain and eventually presented to the user. The Leon wrapper implementation is almost as short as Swim wrapper's and we are similarly able to handle arbitrary number of simultaneous Leon tasks. We also implemented the *finish* method that is responsible for stopping Leon tasks.

6.4.5 Swim Wrapper

Our implementation of the Swim wrapper supports synthesis from examples. Once the user selects a csv file with examples in the GUI, the generated job with essential data is sent to the Swim wrapper. A new process is created by the wrapper to handle that request. The wrapper reads received data which is the name of a task and a path to the csv file, and builds a proper command line expression to be executed. The command is responsible for communication with the jar file of the Swim library that is stored in the directory /opt/psynth/swim/. As the Swim computation is running, the output appears in a temporary file which is monitored in the real time by the wrapper in the /tmp/psynth/swim/ directory (implemented analogously as described in Section 6.4.5). Obtained results are parsed (see Section 6.4.2) and returned to the upper levels of our system (those inaccessible to user) and displayed in the GUI. The implementation of the Swim wrapper also includes the code responsible for killing a particular Swim process if such an intent is communicated by the user (see Section 6.4.3).

It is worth noting that due to the fact that every Swim job is executed as a separate process, our system is capable of handling many Swim job requests at the same time. Swim wrapper gives each job a unique id which is then used in the names of output files

to guarantee that jobs do not interfere with each other.

The whole Swim wrapper implementation is done in less than 150 lines of code.

6.5 Tests

Software testing is used to test the quality of computer programs and plays an important role in software development life cycle. There are many techniques of testing programs and there is no *'one size fits all'* solution to the problem.

Tests could be classified into two categories:

- *manual* - those kinds of tests usually require little to no setup and closely resemble how the end user is going to use the software product. Their drawbacks are that they may be error prone (for example because of tester being tired) and they have to be executed each time by a human (which takes considerable amounts of time).
- *automatic* – they take some time to set up, experienced software developer has to create them and they need to be updated often to be up to date. Running them usually requires no interaction with the user and they tend to be very fast and reliable.

There has to be a balance between time spent on writing programs and testing them. We took 3 significant aspects into consideration when choosing our testing techniques:

- time spent on writing tests and/or running tests
- accuracy of tests (meaning how many bugs are they likely to catch)
- transparency when a test fails – some testing techniques can exactly pinpoint where is the problem (unit tests), while others just say that there is a problem and programmer has to do further inspection to find the cause of the problem (functional tests).

With that in mind, we decided to use 2 different approaches of testing software in our project:

- automatic unit tests
- manual acceptance tests

6.5.1 Unit tests

We used unit tests where it was easy to write them and where the code that we wanted to test was not so straightforward to implement. In the end, we wrote unit tests for three different functionalities:

1. Extracting color information from program output to be able to show eye-catching results like this:
2. Parsing Leon verification output to extract results for each method (see section 6.4.2).

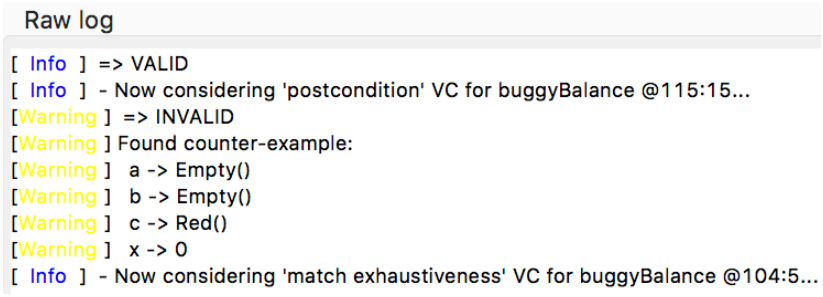

Figure 6.10: Colorful output of a Leon verification task.

3. Fallback (when AST fails) function that extracts method names from a code fragment (see section 6.1.5.1).

To give interested reader an idea of how our unit tests looked, we are going to show how we tested the first of the functionalities listed above. We prepared a couple of carefully selected inputs that use ANSI codes (see the 6.1.5 section) like this:

1. '[[31m[1m Fatal [0m] There were errors.' (which translates to '[**Fatal**] There were errors.' when you see that in a terminal)
2. '[[34m Info [0m] [1mLeon verification and synthesis tool (http://leon.epfl.ch/)[0m\n [[34m Info [0m] '

With that input we tested:

- Removing ANSI codes from text. This is necessary because StyledText widget that we used to present text to the user cannot handle ANSI codes on it's own. If we do not remove those codes, they will be presented to the user like this:

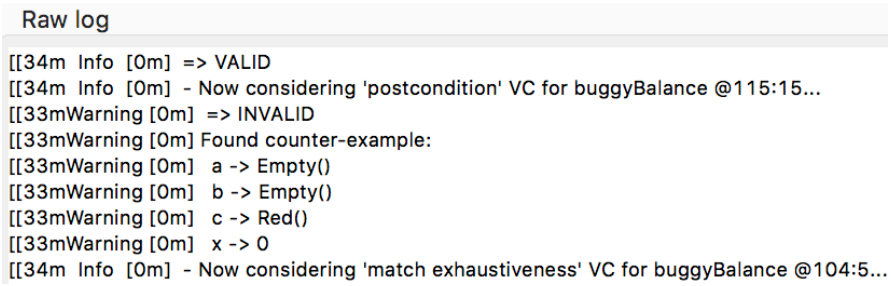

Figure 6.11: Log without ANSI escape codes bytes removed.

We checked that our methods correctly removed ANSI codes bytes - for example for input 1) expected result is: '[Fatal] There were errors.'

- Getting style information - we checked that our method correctly returns positions of styled text with information about the style type. For example for input 1) expected result is information that from position 1 to 8 text should be red and bolded. That style information is used in the dialog with task results to format StyledText widgets.

6.5.2 Testing different Eclipse versions

We checked that our plugin is working on Eclipse Neon 4.6 (both neon.1 and neon.2). With small modifications to MANIFEST.MF file (changed required versions for packages) we were also able to successfully run our plugin in Eclipse Mars 4.5.

Chapter 7

Experiments

Our project is designed to provide integration of a wide-range of external libraries for searched based software engineering within the Eclipse environment. As a proof of concept, the current version of our project delivers out of the box integration with two such libraries - Leon and Swim. As a part of this thesis, we are interested in how such libraries perform in practice and what real influence may they have on developers' everyday's programming routine. Therefore, we designed a series of experiments to test Leon and Swim libraries capabilities under various conditions and discuss their potential.

7.1 Experimental setup

We use Leon library (version 3.0, commit SHA 070e74d6) and Swim library (commit SHA c29d270) through our own Eclipse perspective. We test them for features that are supported by our plugin i.e. verification and synthesis from specification for Leon and synthesis from examples for Swim. For each of these functionalities we prepare relevant test cases. Some of them are adopted from test cases provided by the creators of those libraries and some of them are generated by us. Since both libraries offer mutually exclusive features and work under different paradigms (Leon uses exact methods whereas Swim uses approximate methods), it is impossible to objectively compare their performance with each other and therefore test for each library are mostly independent of test for the other library. However, by preparing Leon tests for synthesis from specification in the example-like manner, we try to relate it to specification from examples offered by Swim.

All experiments in this chapter were run on a computer with Intel Core i5-5200U processor, 16GB 1600MHz RAM, Samsung 850EVO SSD and Ubuntu 15.10 operating system.

7.2 Test cases

7.2.1 Leon verification

We selected several test cases written in Scala that could occur in the real life during software development. They include:

- Red-Black Tree from Leon's examples [45] - correct and faulty implementation
  Red-Black Tree is implemented with 13 methods:
  - *content* - shows the content of a tree as a set
  - *size* - returns the size of a tree
  - *isBlack* - checks if the given node is black
  - *redNodesHaveBlackChildren* - checks if red nodes have black children in a whole tree
  - *redDescHaveBalckChildren* - checks if descendants of red nodes in a tree have black children
  - *blackBalanced* - checks if a tree is balanced
  - *blackHeight* - returns the height of a tree with respect to black nodes
  - *ins* - insert a given element into a tree
  - *makeBlack* - make the root of a given tree black
  - *add* - add a given element to a tree (uses ins and makeBlack)
  - *balance* - balances a given tree
    Incorrect implementation includes the following methods:
  - *buggyAdd* - incorrect version of add, missing the condition that a tree is balanced in both precondition and postcondition, omitted makeBlack in the body of a function
  - *buggyBalance* - incorrect version of balance, missing one case of the balancing algorithm

  All methods except for buggyAdd and buggyBalance have been verified as valid.
  For buggyAdd the following counter-examples were found:

```
(postcondition) t -> Node(Red(), Empty(), 0, Node(Black(), Empty(),4, Empty()))
x -> 0
(precondition) t -> Node(Black(), Empty(),0, Node(Black(), Empty(),0, Empty()))
```

  For buggyBalance the counter-example is:

```
a -> Empty()
b -> Empty(
c -> Black()
x -> 0
```

  Verification time: 32 seconds
- Insertion Sort from Leon's examples [46] - correct and faulty implementation
  This implementation is an Insertion Sort object, from which we selected 7 methods. The methods are:
  - *size* - returns the size of a given list
  - *contents* - returns the content of a list as a set

  - *isSorted* - checks whether a given list is sorted
  - *sortedIns* - inserts an element into a sorted list so that remains sorted
  - *sort* - sorts a given list
  - *mergeInto* - merges one list which is not sorted into a different sorted list
    Incorrect implementation includes the following methods:
  - buggySortedIns - the incorrect version of the method sortedIns, the precondition that an existing list is sorted is missing

  The method buggySortedIns has been verified invalid by postcondition and the counter-example is:

```
e -> 449
l -> Cons(450, Cons(-1791, Nil()))
```

  Verification time: 19 seconds

- Amortized Queue from Leon's examples [47] - correct implementation
  This object consists of 14 methods. They are:

  - *size* - returns the size of a given list (we canceled verification by postcondition for the size method after 62 minutes)
  - *content* - returns the content of a list as a set
  - *asList* - returns the given queue as a list
  - *concat* - concatenates two given lists
  - *isAmortized* - checks if the queue is amortized
  - *isEmpty* - checks if a queue is empty
  - *reverse* - returns a given list in a reversed order
  - *amortizedQueue* - returns an amortized queue created from two given lists
  - *enqueue* - adds an element to a queue
  - *tail* - returns the tail of a queue
  - *front* - returns the front of a queue
  - *propTail* - checks that tail method is working correctly
  - *enqueueAndFront* - checks that enqueue and front methods are working correctly together
  - *enqueueDequeueTwice* - checks that enqueue, tail and front methods are working correctly together

  Verification results: all methods were verified as valid

  Verification time: 60 seconds (not counting the 62 minutes for canceled verification of the size method)

7.2.2 Leon synthesis from specification

- List Insert from Leon's examples [48]

It provides an incomplete code for the list insertion. The function to be synthesized is called insert. It is supposed to accept an integer and insert it into a given list. This specification is given as a choose structure but the implementation is missing.

Code:

```
object Insert {
  sealed abstract class List
  case class Cons(head: BigInt, tail: List) extends List
  case object Nil extends List

  def size(l: List) : BigInt = (l match {
    case Nil => BigInt(0)
    case Cons(_, t) => 1 + size(t)
  }) ensuring(res => res >= 0)

  def content(l: List): Set[BigInt] = l match {
    case Nil => Set.empty[BigInt]
    case Cons(i, t) => Set(i) ++ content(t)
  }

  def insert(in1: List, v: BigInt) = {
    choose { (out : List) =>
        content(out) == content(in1) ++ Set(v)
    }
  }
}
```

Synthesis results:

```
def insert(in1 : List, v : BigInt): List =  {
  Cons(v, in1)
} ensuring {
  (out : List) => content(out) == content(in1) ++ Set[BigInt](v)
}
```

Synthesis time: 23 seconds

- List Union from Leon's examples [49]

  This case partially implements the union of two given lists. The essential method, union, is given without the proper body and the choose structure gives the specification of its behavior - the resultant list should include elements from both of the given lists.

  Code:

```
object Union {
  sealed abstract class List
  case class Cons(head: BigInt, tail: List) extends List
  case object Nil extends List
```

```
  def size(l: List) : BigInt = (l match {
    case Nil => BigInt(0)
    case Cons(_, t) => 1 + size(t)
  }) ensuring(res => res >= 0)

  def content(l: List): Set[BigInt] = l match {
    case Nil => Set.empty[BigInt]
    case Cons(i, t) => Set(i) ++ content(t)
  }

  def union(in1: List, in2: List) = {
    choose { (out : List) =>
        content(out) == content(in1) ++ content(in2)
    }
  }
}
```

Synthesis results:

```
def union(in1 : List, in2 : List): List =  {
  in1 match {
    case Nil =>
        in2
    case Cons(head, tail) =>
        Cons(head, union(tail, in2))
  }
}
```

Synthesis time: 25 seconds

- UnaryNumerals Add from Leon's examples [50]

  The method add is supposed to add two numbers which are given in a standard or unary representation and return the result. The choose structure captures this specification and the implementation itself should be synthesized.

  Code:

```
object Numerals {
  sealed abstract class Num
  case object Z extends Num
  case class  S(pred: Num) extends Num

  def value(n: Num): BigInt = {
    n match {
      case Z => BigInt(0)
      case S(p) => 1 + value(p)
    }
  } ensuring (_ >= 0)

  def add(x: Num, y: Num): Num = {
    choose { (r: Num) =>
```

```
        value(r) == value(x) + value(y)
    }
  }
}
```

Synthesis results:

```
def add(x : Num, y : Num): Num =  {
  x match {
 case Z =>
  y match {
    case Z =>
      Z
    case S(pred) =>
      S(pred)
  }
 case S(pred) =>
  S(add(pred, y))
  }
}
```

Synthesis time: 25 seconds

7.2.3 Swim synthesis from examples

To automate our experiments as much as possible, we created a Python script that given a function (that takes any number of boolean arguments and returns a boolean value) creates test files (with selected by user number of examples chosen randomly from all possible inputs) both for Swim and Leon. Therefore all functions listed below will be presented using Python syntax, so that the reader can use our script to recreate the inputs, possibly slightly change a function and run experiments himself. In all of our experiments we used a constraint for maximum number of generations = 50.

Our first function that we run experiments used 6 input variables, and looked like this:

```
def fun(a: bool, b: bool, c: bool, d: bool,e: bool,f: bool)->bool:
 return ((not (a or b)) and ( d or not (c and f))) or e
```

Swim had no trouble finding results with passed tests ratio =1, no matter what set of inputs we chose.
For all possible 64 inputs results are:

- 1.9 seconds to complete
- best generation = 11
- found function:

```
|(4 !|(|(|(1 !|(3 !&(|(1 |(3 5)) !&(|(0 5) !|(2 0))))) 0) !|(!&(5 4)
    |(!&(!|(!|(|(2 1) |(1 2)) 0) 4) &(!|(3 !|(!|(5 3) 1)) 5)))))
```

- passed test ratio = 1

For randomly selected 13 inputs, results are:

- 0.6 seconds to complete
- best generation = 0 (the first one)
- found function:

```
|(|(4 &(&(4 1) !&(3 5))) !|(&(&(2 0) !&(5 5)) 1))
```

- passed test ratio = 1

We then tried to find a set of examples, that Swim would have a hard time to find a function that satisfies all of them. That lead us to this function:

```
def fun(a: bool, b: bool, c: bool, d: bool,e: bool,f: bool,g: bool,h: bool)->bool:
 return (g and f or not h) and ((a and b) or ( c and not f) or ((not d and e) and
     (not a and f)))
```

Firstly, we generated all 256 possible inputs. Swim results are:

- 8.5 seconds to complete
- best generation = 21
- found function:

```
!&(!&(2 !|(7 &(2 |(5 |(5 7))))) !&(!&(7 !|(!&(|(!&(&(!&(4 3) |(5 5)) 6) &(2
    2)) 7) |(!&(|(7 0) 1) !|(&(!&(|(|(|(2 2) 1) &(!&(6 5) 4)) !&(!|(!&(4 1)
    !|(0 2)) !&(|(0 1) 4))) 1) !&(6 5))))) &(0 1)))
```

- passed test ratio = 0.953 (12 failed tests)

Secondly, we chose randomly only 62 inputs. Swim results are:

- 4.7 seconds to complete
- best generation = 38
- found function:

```
!|(!|(!|(!|(1 2) &(7 !&(!|(!|(6 6) 7) 5))) &(5 !|(!&(&(1 |(!&(&(5 !|(6 6))
    !|(|(7 3) !|(1 0))) !&(1 3))) |(!|(!|(|(7 6) !|(2 1)) 0) 6)) !|(|(|(&(4
    4) 1) &(6 0)) !&(!&(&(5 2) 4) 6))))) !|(&(6 !|(0 &(!&(0 !|(&(0 5) !|(0
    7))) !|(!|(&(0 4) !&(2 0)) !|(!|(2 0) 7))))) |(&(&(4 4) !&(4 3)) 0)))
```

- passed tests ratio = 0.984 (1 test failed)

Last, but not least we created a set of contradictory examples for which no function exists. For this we used function:

```
def fun(a: bool, b: bool, c: bool, d: bool,e: bool)->bool:
 return not ((a and c) and (b or not e)) #d not used on purpose
```

and added to the 32 generated examples one more:

```
true false true true true false (our function returns for this set of input values
     result true)
```

Swim results are:

- 4.7 seconds to complete
- best generation = 2
- found function:

```
&(|(!|(&(|(|(!|(1 4) 3) !&(!|(0 3) &(1 2))) &(!&(!&(4 0) &(1 2)) |(&(3 3) 1)
    )) !|(|(4 2) 3)) !&(&(1 &(2 2)) 2)) !&(!|(!|(2 0) !&(2 0)) |(!&(4 0)
    !|(0 3))))
```

- passed test ratio = 0.967 (1 failed test)

7.2.4 Swim and Leon synthesis from examples - comparison

We created a function defined like this:

```
def fun(a: bool, b: bool, c: bool, d: bool,e: bool) -> bool:
 return ((a and b) or not c) or (not d and e)
```

We generated all possible inputs (there are 2^5 of them) and we choose randomly 9 of them. We run our function on those selected inputs to get a return value. We then created test input CSV file for Swim and a test object for Leon:

```
import leon.lang._
import leon.lang.synthesis._
import leon.collection._
object Test {
  def examples(a0 : Boolean, a1 : Boolean, a2 : Boolean, a3 : Boolean, a4 :
      Boolean) =
  {
    ???[Boolean]
  } ensuring { (res: Boolean) =>
    ((a0, a1, a2, a3, a4), res) passes {
      case (true, true, true, false, true) => true
      case (true, false, false, false, false) => true
      case (true, false, true, false, false) => false
      case (false, true, false, true, false) => true
      case (true, false, true, false, true) => true
      case (false, false, true, false, false) => false
      case (false, false, true, true, false) => false
```

```
      case (false, false, false, false, false) => true
      case (false, true, false, true, true) => true
    }
  }
}
```

Both Leon and Swim succeeded in finding a function that satisfied those examples. The original function was not found, because since we only provided 9 out of 32 possible inputs there were many functions that satisfy those requirements. Swim's results:

- found function:

  ```
  !&(!|(!&(!&(3 0) !&(2 4)) 4) 2)
  ```

- 1.5 seconds to complete
- success rate=1 (meaning no failed tests)

It took Leon 4 minutes and 13 seconds to come up with:

```
if (a1) {
  true
} else {
  a2 == a4
}
```

In all of the examples in this section we had to help Leon with choosing rules, because the program would hang forever on Symbolic Term Exploration at the start. After canceling this rule program then proceeded to use a combination of ADT Splits mixed up with Symbolic Term Exploration. We also had to run Leon with –ste:maxsize=5 argument, otherwise Leon was creating 9 hardcoded if statements:

```
def examples(a0 : Boolean, a1 : Boolean, a2 : Boolean, a3 : Boolean, a4 : Boolean)
    : Boolean =
{
  choose((hole$1 : Boolean) => ((a0, a1, a2, a3, a4), hole$1) passes {
    case (true, true, true, false, true) =>
    true
    case (true, false, false, false, false) =>
    true
    case (true, false, true, false, false) =>
    false
    case (false, true, false, true, false) =>
    true
    case (true, false, true, false, true) =>
    true
    case (false, false, true, false, false) =>
    false
    case (false, false, true, true, false) =>
    false
    case (false, false, false, false, false) =>
```

```
    true
    case (false, true, false, true, true) =>
    true
  })
}
```

After we generated all 32 possible outputs for our function, Swim found solution:

- found function:

```
!&(&(!&(!&(3 3) &(!&(3 3) |(3 4))) !&(|(!&(0 !|(1 2)) 1) &(!&(|(|(!|(4 1) &(3
    0)) &(4 &(|(3 2) 0))) !&(0 2)) !|(!|(1 1) !|(0 0))))) &(|(|(|(!|(1 3)
   &(2 1)) |(0 |(1 3))) !|(!&(4 |(1 1)) 2)) 2))
```

- 4 seconds to complete
- success rate=1 (meaning no failed tests)

Leon was not able to produce results in a given time frame of 30min.

The primary difference between synthesis from examples using Leon and Swim is that Leon gives you guarantees about synthesised function being correct, while Swim might return a function that fails on some cases (see subsection above). In examples provided in this section Swim was much faster and finished with success rate=1.

7.2.5 Discussion

We tried to use test cases for experiments that resemble real life scenarios as much as possible. Majority of the experiments were solved quickly (order of seconds) and provided valuable information to the user. Several slow cases were identified (see Leon synthesis from examples 7.2.4) and cases where tools were not able to either provide a 100% correct result (Swim when the synthesized function failed on several examples) or to provide a result at all (Leon - synthesis from examples and verification of AmortizedQueue.size method).

From the user's perspective using Swim is very straightforward. On the other hand, using Leon has in our opinion two slight inconveniences. Firstly, it is sometimes difficult to come up with required assertions (require, ensuring) that Leon uses. Users might be tempted to use (well known to them) unit tests instead to try to guarantee correctness of their programs. Secondly, Pure Scala (as for now) is really limited, and it is going to be challenging, to say the least, to use Leon when working with multiple libraries and modern frameworks that are not limited to Pure Scala.

The first argument is not even Leon's fault and it is just a matter of users getting used to tools like Leon. As for the second one, we are sure that in the future restrictions (on allowed language constructs) will be gradually lifted. Moreover, as shown in the examples, there are countless possibilities to use Leon as it is now and it is up to the developers to show their skills and make it work. All in all, we conclude that the Search Based Software Engineering techniques delivered through our plugin can be successfully and efficiently used by programmers for many common problems they encounter while writing code.

Chapter 8

Final remarks

By implementing our plugin, we created a bridge between the ideas and tools that scientists gave birth to and common tools available to software developers in the industry. Thus, we brought the concept of the Search Based Software Engineering from the academic to the real-life setting. We believe that the acceleration of the development of the Search Based Software Engineering requires combined efforts from both of these environments and therefore we prepared our plugin in the widely-open manner by caring about the easiness of its extendability through custom wrappers that can be prepared with low effort for every external library.

Having performed experiments of selected libraries that are incorporated into our plugin, we conclude that the Search Based Software engineering is a very promising concept. Although the field is yet very young and its capabilities are limited by the processing power and available techniques, as we proved, it is already possible to use its accomplishments in practice and in reasonable time. They significantly enhance the process of software development by assisting programmers in small tasks, which are quite common, and despite their innocent size often turn out to be tricky and time-consuming like, for example, predicting all of the possible edge cases for which the method can fail and thus should be thoroughly tested.

8.1 Goals accomplished

By comparing our initial objectives from the section 1.2 and actual developments of our work described in this thesis, we conclude that all of the objectives have been met.

We delivered a fully functional plugin for the Eclipse IDE which supports Search Based Software Engineering techniques from external libraries. In particular, we achieved the following goals (compare with the list in the section 1.2):

- Our system is designed in a highly generic way so that many external libraries can be incorporated into it with only little effort of the user (idea of wrappers).
- Results of a computation are delivered real-time to the user both in raw and parsed versions.
- Our system works on Unix-like systems, in particular it was tested on recent distributions of Linux (Ubuntu) and macOS.

- Functionalities offered by our plugin can be initialized by only few clicks in specially prepared menus in our plugin's perspective and terminated any time by the user.
- Our plugin delivers out-of-the-box support for Swim and Leon external libraries which provide tools for synthesis from examples, synthesis from specification and verification of software.
- Universal architecture of our system was described in details in this thesis and we believe that our approach and know-how could be used by other systems which aspire to foster universality and extendability in the sense of incorporation of external tools.

8.2 Potential extensions

In this section we present number of potential improvements and extensions that are in our opinion noteworthy.

Windows support

Windows is one of the most popular operating system. Over 50% of developers declared using Windows as their primary development OS in the survey [22]. Thus we believe that it would be vital to provide this integration.

Parametrizing orders

It would be very convenient for the users to have an opportunity to parametrize orders accordingly to their preferences from the level of IDE. For instance, in GP user would be able to select the maximum number of generations, or choose the specific selection operator. It would require to extend the current capabilities of the IWrapper class, as well as to implement the specific graphical elements allowing to parametrize the orders accordingly to the provided options.

Remote task execution

While user is using Eclipse with our plugin on his personal computer, tasks could be delegated to run on a different machine. This could not only greatly reduce CPU load on user's personal computer but also speed up execution of tasks. Remote machine could be a much more powerful computer that could compute results multiple times faster. Apart from that, this would remove requirements of installing external libraries like Leon and Swim on user's personal computer. It would be enough to install those libraries only on the remote server and multiple users could be using the same server.

Some users might be interested in checking out our plugin, but installing all of the required libraries (Swim, Leon) might be too much of an effort for them to justify. We could offer a server with installed Swim and Leon libraries that the user could use for a short period of time, while he is getting to know our plugin.

UX improvements

Readability of the Task List could be slightly improved. For example, an alternative Task List could look somewhat like this:

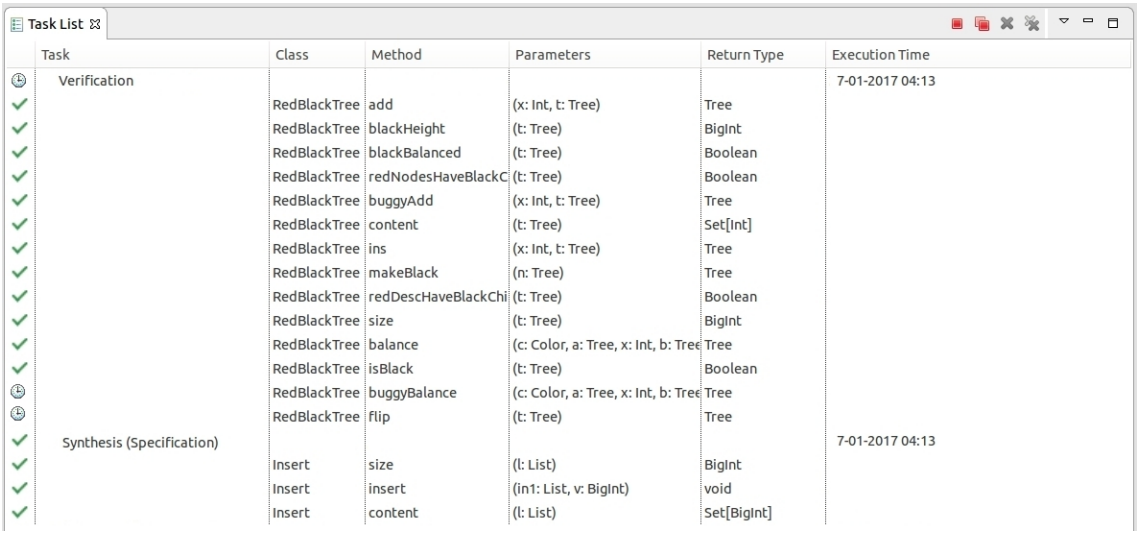

Figure 8.1: Improved Task List.

This way, it would clear for the user that a task consists of many subtasks, and stopping one of the subtasks (for example synthesis of a particular method) means stopping all of the other subtasks that belong to the same group. Also, it would be nice if user could collapse a task group to hide some of the information that he might not be interested in in a particular moment. Another feature that would be worthwhile is reporting status on a per subtask level (buggyBalance and flip are still being verified while verification of other methods has finished).

Appendix A

User Guide

A.1 Installing steps

Currently, our plugin is supported on GNU/Linux and macOS. To install the plugin follow these steps:

1. Use Eclipse Neon (4.6).
2. Install 'Scala IDE 4.2.x' plugin from Eclipse Marketplace to your eclipse IDE.
3. Download source code repository of our plugin.
4. Download our exported plugin (psynth.jar) and wrappers: LeonWrapper.jar and SwimWrapper.jar.
5. To be able to use Leon:
   - 5.1. on macOS install GNU sed (gsed)
   - 5.2. run `scripts/modify_scalaide_core_jar.sh /path/to/your/eclipse/directory` (script can be found in project's source code root directory)
   - 5.3. install Leon (follow steps from http://leon.epfl.ch/doc/installation.html)
   - 5.4. make Leon command systemwide executable - this can be done for example by creating a symbolic link to your Leon script in /usr/local/bin directory (run `sudo ln -s /path/to/your/leon/executable_script /usr/local/bin/leon`)
6. To be able to use swim:
   - 6.1. install git, scala and sbt
   - 6.2. on macOS also install GNU sed (gsed)
   - 6.3. run `scripts/set_everything_up.sh` (script can be found in project's source code root directory)
     - 6.3.1. you will be asked for your password
     - 6.3.2. this command will get latest versions of Fuel and Swim libraries, build those libraries and put them in /opt/psynth directory. This was working as of 31.01.2017. Since then authors of those libraries might have chosen a completely different approach of building their software (for example different project structure, new dependencies etc.). In that case either use older version of those libraries or build those libraries yourself and put

swim.jar and leon.jar in /opt/psynth/swim

7. Put downloaded wrappers of your choosing in /opt/psynth/wrappers directory.
8. Make sure you have access rights to /opt/psynth and all nested directories.
9. Put downloaded psynth.jar in your eclipse plugins directory.
10. Close eclipse and restart it with eclipse -clean command.
11. Open program synthesis perspective.

A.2 Use cases

In this section we present typical use cases of our plugin.

(a) Swim synthesis from examples

1. The user opens Eclipse environment.
2. The user creates or opens a project.
3. The user opens our plugin's perspective.
4. The user selects 'Swim' from the 'Library' section of the top toolbar.
5. The user selects 'Synthesis examples' from the 'Action' section of the top toolbar.
6. The user selects a csv file (or multiple files) with pre-prepared examples.
7. The user waits for a task to be marked as finished.
   (A) The user can see live progress if he chooses to do so (same as VIII).
8. The user double-clicks the relevant task entry in the Task List.
9. The user analyzes raw and parsed output.
10. The user copies a generated code and pastes it into a relevant place in his project.

(b) Leon verification

1. The user opens Eclipse environment.
2. The user creates or opens a project.
3. The user opens our plugin's perspective.
4. The user selects 'Leon' from the 'Library' section of the top toolbar.
5. The user selects 'Leon' from the 'Library' section of the top toolbar.
6. The user opens a relevant file with a Scala code in the editor and optionally highlights selected methods.
7. The user selects 'Verification' from the 'Action' section of the top toolbar.
8. The user waits for a task to be marked as finished.
9. The user double-clicks the relevant task entry in the Task List.
10. The user analyzes raw and parsed output, in particular, result of verification and potential counterexamples.

(c) Leon synthesis from specification

1. The user opens Eclipse environment.
2. The user creates or opens a project.
3. The user opens our plugin's perspective.
4. The user selects 'Leon' from the 'Library' section of the top toolbar.
5. The user opens a relevant file with a Scala code in the editor.
6. The user prepares methods with specification within the 'ensuring' structure.
7. The user selects 'Synthesis specification' from the 'Action' section of the top toolbar.
8. The user waits for a task to be marked as finished.
9. The user double-clicks the relevant task entry in the Task List.
10. The user analyzes raw and parsed output, in particular, the synthesized code.

Bibliography

[1] R. S. Boyer and J Strother Moore. Program verification. https://www.cs.utexas.edu/users/boyer/jar.pdf.

[2] A. Gupta. Program verification. http://www.cs.princeton.edu/courses/archive/spr16/cos217/lectures/24_ProgramVerif.pdf.

[3] Krzysztof Krawiec. Program synthesis. http://www.cs.put.poznan.pl/kkrawiec/wiki/uploads/Zajecia/ProgSynthSlides.pdf.

[4] Piotr Stańczyk. *Algorytmika praktyczna: Nie tylko dla mistrzów*. Wydawnictwo Naukowe PWN, 2009.

[5] Saemundur O Haraldsson and John R Woodward. Genetic Improvement of Energy Usage is only as Reliable as the Measurements are Accurate. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 821–822. ACM, 2015.

[6] William B Langdon. Genetic improvement of programs. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2014 16th International Symposium on*, pages 14–19. IEEE, 2014.

[7] Managing Network Usage - Android documentation. https://developer.android.com/training/basics/network-ops/managing.html.

[8] Optimizing Server-Initiated Network Use - Android documentation. https://developer.android.com/topic/performance/power/network/action-server-traffic.html.

[9] Martin Fowler and Kent Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[10] Harry Percival. *Test-Driven Development with Python*. O'Reilly Media, Inc., 2014.

[11] Justyna Petke, Mark Harman, William B Langdon, and Westley Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In *European Conference on Genetic Programming*, pages 137–149. Springer, 2014.

[12] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon Documentation - Repair. http://leon.epfl.ch/doc/repair.html.

[13] BarraCUDA Fast Short Read Aligner. https://sourceforge.net/projects/seqbarracuda/.

[14] William B Langdon, Albert Vilella, Brian Yee Hong Lam, Justyna Petke, and Mark Harman. Benchmarking genetically improved BarraCUDA on epigenetic methylation NGS datasets and nVidia GPUs. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion*, pages 1131–1132. ACM, 2016.

[15] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming*. Lulu. com, 2008.

[16] Melanie Mitchell. *An introduction to genetic algorithms*. publisher, 1998.

[17] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187, 1985.

[18] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[19] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. Scala Language Specification. https://www.scala-lang.org/files/archive/spec/2.12/.

[20] M. Odersky. Scala documentation - What is Scala? https://www.scala-lang.org/what-is-scala.html.

[21] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger. Overview of the Scala Programming Language. http://www.scala-lang.org/docu/files/ScalaOverview.pdf.

[22] Stackoverflow Developer Survey 2016. http://stackoverflow.com/research/developer-survey-2016.

[23] Eclipse IDE download page with statistics about number of downloads. https://www.eclipse.org/downloads/eclipse-packages/.

[24] Rowan Wilson. The Eclipse Public License - An Overview. http://oss-watch.ac.uk/resources/epl.

[25] Eclipse wiki page about Simultaneous Release. https://wiki.eclipse.org/Simultaneous_Release.

[26] Vogella GmbH. Eclipse IDE Plug-in Development: Plug-ins, Features, Update Sites and IDE Extensions. http://www.vogella.com/tutorials/EclipsePlugin/article.html.

[27] The Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/.

[28] Vogella GmbH L. Vogel. OSGi Modularity - Tutorial. http://www.vogella.com/tutorials/OSGi/article.html.

[29] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon documentation. https://leon.epfl.ch/doc/index.html.

[30] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon documentation - Pure Scala. https://leon.epfl.ch/doc/purescala.html.

[31] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon documentation - verification. https://leon.epfl.ch/doc/verification.html.

[32] M. Antognini, R. Blanc, S. Gruetter, L. Hupel, E. Kneuss, M. Koukoutos, V. Kuncak, R. Madhavan, S. Stucki, and P. Suter. Leon documentation - synthesis. https://leon.epfl.ch/doc/synthesis.html.

[33] CVC4 Solver website. http://cvc4.cs.nyu.edu/web/.

[34] N. Bjørner Microsoft Research L. de Moura. Z3 solver presentation. http://research.microsoft.com/en-us/um/redmond/projects/z3/Z3_System.pdf.

[35] Satisfiability modulo theories. https://en.wikipedia.org/wiki/Satisfiability_modulo_theories.

[36] Krzysztof Krawiec. Functional Evolutionary Algorithms. https://github.com/kkrawiec/fuel.

[37] Krzysztof Krawiec. Synthesis with Metaheuristics - Genetic Programming in Scala. https://github.com/kkrawiec/swim.

[38] The Eclipse Foundation. Eclipse Perspectives. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.user%2Fconcepts%2Fconcepts-4.htm.

[39] S. Scholz Vogella GmbH L. Vogel. AJFace Data Binding - Tutorial. http://www.vogella.com/tutorials/EclipseDataBinding/article.html.

[40] S. Scholz Vogella GmbH L. Vogel. SWT - Tutorial. http://www.vogella.com/tutorials/SWT/article.html.

[41] Phan Cong Vinh, Vangalur Alagar, Emil Vassev, Ashish Khare (Eds.). Context-Aware Systems and Applications. page 171, 2013.

[42] Eli Bendersky. Abstract vs. Concrete Syntax Trees . http://eli.thegreenplace.net/2009/02/16/abstract-vs-concrete-syntax-trees/#id5.

[43] Termination Signals man page. http://www.gnu.org/software/libc/manual/html_node/Termination-Signals.html.

[44] Linux Filesystem Hierarchy - /tmp directory. http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/tmp.html.

[45] EPFL Leon. RedBlackTree example. https://github.com/epfl-lara/leon/blob/v3.0/testcases/verification/datastructures/RedBlackTree.scala.

[46] EPFL Leon. InsertionSort example. https://github.com/epfl-lara/leon/blob/v3.0/testcases/verification/datastructures/InsertionSort.scala.

[47] EPFL Leon. AmortizedQueue example. https://github.com/epfl-lara/leon/blob/v3.0/testcases/verification/datastructures/AmortizedQueue.scala.

[48] EPFL Leon. Insert example. https://github.com/epfl-lara/leon/blob/master/testcases/synthesis/current/List/Insert.scala.

[49] EPFL Leon. Union example. https://github.com/epfl-lara/leon/blob/master/testcases/synthesis/current/List/Union.scala.

[50] EPFL Leon. Numerals example. https://github.com/epfl-lara/leon/blob/master/testcases/web/synthesis/18_UnaryNumerals_Add.scala.