Towards a GUI for Program Verification with KeY

Master of Science Thesis in the Programme Software Engineering and Technology

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, January 2010

Mohammad Ali Darvish Darab

Examiner: Reiner Hähnle

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden 2009

Contents

1 Introduction **3**
- 1.1 Outline . . . 4

2 Background **6**
- 2.1 KeY System . . . 7
- 2.2 JavaCard Dynamic Logic, Sequent Calculus, and Taclets . . . 8
- 2.3 Updates . . . 10
- 2.4 Symbolic Execution . . . 11
- 2.5 Design By Contract . . . 13
- 2.6 Formal Specification . . . 14
  - 2.6.1 Object Constraint Language . . . 14
  - 2.6.2 Java Modeling Language . . . 15

3 Visual Symbolic Execution Debugger **19**
- 3.1 What VSED is . . . 19
- 3.2 Execution Tree . . . 20
- 3.3 Execution Tree View . . . 20
- 3.4 Statement Breakpoints . . . 21
- 3.5 Symbolic State Visualization . . . 22

4 Short Evaluation Of VSED **26**
- 4.1 Quicksort . . . 26
- 4.2 Mergesort . . . 31

5 Towards Finite Execution Trees **38**
- 5.1 Loop Invariant . . . 38
- 5.2 Loop Invariant Rule . . . 39
- 5.3 Loop Invariant Rule Visualization . . . 39

6 Modular and Project-based Verification **42**
6.1 Project-based Verification 42
6.1.1 Eclipse IDE 42
6.1.2 KeY Project 44
6.2 Modular Verification 46

7 Conclusion **49**
7.1 Contributions 49
7.2 Conclusions 49
7.3 Future Work 50

Chapter 1

Introduction

This work contributes to a new prototypical Graphical User Interface (GUI) aimed for visual formal proving of Java programs using KeY[1] prover. KeY is an interactive formal prover based on symbolic execution that uses an instance of dynamic logic to reason about Java programs. KeY, by executing programs symbolically, enables users to do interactive proofs as the program gets executed. Symbolic execution is a technique in which instead of supplying concrete values as program input, symbolic values are used in order to substitute all the concrete values that take an unique execution path.

This work adds on to an already-existing Visual Symbolic Execution Debugger(VSED)[2] that has already been realized as a plug-in into Eclipse IDE[3] environment. VSED, based on symbolic execution, draws an execution tree view of Java programs and offers debugging functionalities on it. Of course, VSED is not stand-alone; In fact, it runs the KeY prover in background which does the actual symbolic execution, and VSED draws a tree view of the symbolically executed program inside Eclipse, plus that it provides debugging functionalities too. Even though VSED is useful as an omniscient debugger (i.e. a debugger that is able to go backwards in time) and also as a tool to comprehend programs better, it has certain limitations which does prevent it from doing program verification.

The main idea of this work has been to overcome these limitations so that it would become possible to do program verification through Eclipse IDE too. One of the main obstacles has been the way the loops are treated in VSED. Since loops can (potentially) execute for an infinite number of times, it is

[1] http://www.key-project.org
[2] http://www.key-project.org/download/#proofvis
[3] http://www.eclipse.org

not possible to generate finite symbolic execution trees by simply unwinding (executing) them, which is what VSED does with loops. The solution to this is to support loop invariant rule. Loop invariant rule is a logical rule that proves a loop correct utilizing a so-called loop invariant and results in generating a finite execution tree.

However, one might ask why at all having an alternative way (the new GUI) for what already exists (KeY GUI)? The answer is that the main reason behind doing such a work was to have a less complicated GUI in order to make it easier for non-experts or newcomers to the field to perform formal verification. For example, this new GUI hides away pure logical level rules and only concentrates on executable statements. Besides, taking the first steps of having an alternative GUI for KeY inside a prevalent IDE such as Eclipse will likely help to make KeY more accessible in the future.

Finally, while the loop invariant rule support has been the main focus of this work, there are three more contributions that are also made. The first one is an evaluation of VSED which is given in chapter 4. The second one is support of modular verification concept. The main idea of modular verification is to concentrate on verification of one module (i.e. one method) at a time. Moreover, it has also been desirable to have a common verification settings (i.e. KeY settings) for all the modules (methods) that are part of a single unit (i.e. Eclipse Java project).

1.1 Outline

The subsequent chapters of this report are organized as follows:

- Chapter 2 - Background: It gives an overview of the KeY system. It also discusses some basic concepts that a general understanding of them is required to follow the rest of the report (e.g. Dynamic Logic, Sequent Calculus, Symbolic Execution etc).
- Chapter 3 - Visual Symbolic Execution Debugger: It provides an overview of Visual Symbolic Execution Debugger functionalities.
- Chapter 4 - Short evaluation of VSED: It provides two small case studies with VSED to see how it can possibly help better in debugging programs over conventional debuggers.
- Chapter 5 - Towards Finite Execution Trees: It illustrates loop invariant rule and a way of visualizing it in the new GUI.

- Chapter 6 - Modular and Project-Based Verification: It discusses a new type of project introduced in Eclipse IDE called KeY project. Method contract rule and a way to visualize it is also discussed in this chapter.
- Chapter 7 - Contributions, Conclusions and Future Work: It summarize contributions of the work, draws some conclusions, and discusses a number of possible future works.

Chapter 2

Background

The KeY system is the main result of KeY project[1] that is a joint effort of researchers from University of Karlsruhe, Chalmers University of Technology, and University of Koblenz. The KeY system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of software as seamlessly as possible[4]. This states an ultimate goal for KeY project, but here we merely look at KeY as a theorem prover which is used as a verification tool for Java programs.

In fact, KeY system supports a subset of Java programming language called Java Card[2]. Java Card misses a number of constructs of Java language like floating type, char type, threads etc. KeY uses a logic called Java Card Dynamic Logic, which is a typed dynamic logic tailored to Java Card, to reason about programs.

As for specification languages, KeY supports OCL[3] (Object Constraint Language) and JML[4] (Java Modeling Language). KeY uses symbolic execution and Dynamic Logic to verify Java programs. This chapter sets the stage. It provides a brief overview of the KeY system. Then, it introduces some necessary foundational concepts like Symbolic Execution, Dynamic Logic, and JML specification language that are needed for following the subsequent chapters.

[1] http://www.key-project.org/
[2] http://www.java.sun.com/javacard
[3] http://www.omg.org
[4] http://www.cs.ucf.edu/ leavens/JML/

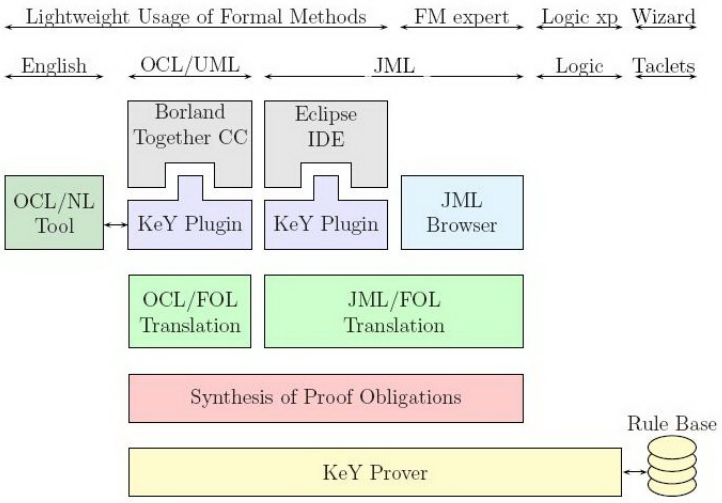

Figure 2.1: Architecture and interface of the KeY system

2.1 KeY System

KeY targets Java Card, which is a subset of Java programming language, and uses an instance of Dynamic Logic to reason about programs written in it. In fact, KeY verifies that programs are doing what they are really intended to do. Expectations from a given program are expressed formally written in a specification language. Specification languages are formal languages that express something about a program at a higher level than the source code. Specifications are not executable themselves. KeY uses symbolic execution to execute programs source code and verify that they comply with their specifications.

Figure 1.1 shows an overall architecture of KeY system[4]:

As it can be seen in Fig 1.1, at the very top level there are CASE-Tools like Eclipse and Borland Together CC where users write their programs and specifications in. At the intermediate level, specifications written in formal specification languages (i.e. JML or OCL) are translated into logic and from that proof obligations are generated. Proof obligations for a given program are generated formulas that are to be proved. And, at the lowest level is KeY prover itself which via interacting with a rule base tries to actually prove formulas. Rules of the rule base are formulated as a lightweight tactics called taclets. A more detailed explanation of taclets is given in the next section.

2.2 JavaCard Dynamic Logic, Sequent Calculus, and Taclets

Dynamic logic is an extension to first order logic. It extends first order logic by a diamond <> and box [] modality. Two basic modalities are possibility and necessity , which are diamond and box modalities, respectively. $< \pi > \phi$ and $[\pi]\phi$ are DL formulas where ϕ is a formula in first order logic and π is a program. These modalities are interpreted as follows:

$< \pi > \phi$ holds iff π terminates and after the execution of π formula ϕ holds.
$[\pi]\phi$ holds iff from the termination of π, it follows that after the execution of π formula ϕ holds.

A sequent is a construct of the form
$\Gamma \Rightarrow \Delta$
where Γ and Δ are formulas in JavaCardDL. Γ is called antecedent and Δ is called succedent. we write rules in the following way:

$$\mathsf{X}\ \frac{\Gamma_1 \Rightarrow \Delta_1 \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma_{\Rightarrow}\Delta_0}$$

where X denotes the rule name. The part below the line is the conclusion and the part above the line is premise, which there can be n (n>0) number of them. The conclusion holds if all the premises hold. A rule without any premises is called an axiom, for example:

$$\mathsf{close}\ \frac{}{\Gamma, \phi \Longrightarrow \phi, \Delta}$$

KeY system is based upon sequent calculus and has a rich set of rules that fall in five categories:

i **Symbolic Execution Rules**:
It includes all the rules that embody symbolic execution of Java statements.

ii **Loop Rules**:
It includes rules related to prove loops. As it was mentioned in the Symbolic Execution section, loops potentially result in many execution

paths. To be able to close loops and being able to reason about them loop rules are needed.

iii **Non-program Rules:**
It includes purely first order logic rules, which as the name implies they are only purely in logical level and does not execute any Java statements.

iv **Method Contract Rules:**
It includes rules that replace a method invocation by a contract.

v **Update Simplification Rules:**
It includes rules that simplify and apply updates to terms and formulas. Updates are discussed in section 2.3.

Taclets are lightweight tactics which are used to formulate rules in a sequent calculus of dynamic logic [4]. Taclets usually contain the following keyword:

```
[\find (f)][\assumes (a)]
[\replacewith (rw)][\add(add)]
```

\assumes imposes a condition on the applicability of the taclet and has a sequent as its parameter. \find looks for a specific pattern in a sequent. \replacewith applies changes by removing something and replacing it with something else, whereas \add only adds something.

The find and replacewith parts of a taclet are all either terms or sequent. Both find and replacewith parts can be arbitrary sequent. Each replacewith is a premise and is called a goal template whereas the find part is the conclusion of the rule. We can define a so-called schema variables which can be instantiated with syntactical constructs based on their types (e.g. a schema variable of type formula can be instantiated by a JavaDL formula). For example [4]:

```
\schemaVariables {
  \formula phi, psi;
}
\rules {
  \close     { \assumes(phi ==> phi) \closegoal };
  \impRight  { \find(==> phi -> psi) \replacewith(phi==>psi) };
  \cut       { \add(\phi ==> ); add (==>phi) };
  \mpLeft    { \assumes(phi ==>) \find (phi -> psi ==>)
  \replacewith(psi ==>) };
}
```

These are four basic rules that are defined as taclets. Two schema variables ϕ and ψ are also defined. "close" rule seeks for the pattern $\phi \Rightarrow \phi$ and whenever it finds one it closes that proof branch. "impRight" seeks for $\Rightarrow \phi -> \psi$ and replaces it with $\phi \Rightarrow \psi$. "cut" rule is a bit different in the sense that it has \add. \add adds something to the sequent. In this case, it adds a formula to both antecedent and succedent of a sequent. The cut rule written in formula notation would be:

$$\mathsf{cut}\ \frac{\Gamma, \phi \Longrightarrow \Delta \quad \Gamma \Longrightarrow \phi, \Delta}{\Gamma \Longrightarrow \Delta}$$

Also, the cut rule is one of those rules that has more than one premise, so it splits a proof branch into two branches. "impLeft" seeks for $\phi -> \psi \Rightarrow$ and replaces it with $\psi \Rightarrow$ only whenever antecedent has a formula ϕ.
More info and further details can be found on[4]

When KeY starts to prove a sequent, it applies appropriate rules on it and usually it results in different branches that form a tree-like structure where each node of this tree is a sequent in itself. This tree is called proof tree and the sequent is completely proved once all the branches of this tree are closed. A particular node n in the proof tree is labeled with the conclusion of a rule and the name X of the applied rule and the children of n are labeled with premises of the rule X. Whenever a branch in proof tree is closed, it means the sequent of its leaf node is labeled with the premise of a axiom of the sequent calculus, and if a branch is not closed we call it an open branch. Although KeY is an interactive prover, it can also automatically apply rules and close branches in proof tree following a so-called strategy.

2.3 Updates

Updates refer to the formalism in KeY to describe the state changes during symbolic execution [11]. The smallest set of updates is given by [3]:

- skip (empty update) The empty update does not change anything from the state.
- l:=v (function update) For all locations l and terms v, function updates are an assignment of the value v to a location l in the heap. v is an arbitrary term, while l is one of the followings:
  - a program variable (a non-rigid unary function)
  - a field access o.a where o is a reference term and a is a field

  - an array access arr[i] where arr is a reference to an array and i is an integer referring to an index.

- for x {U} (quantified update) Where x is a variable and U is an update. All updates for instances of x get done at once.
- if ϕ {U} For all updates U and formula ϕ, the update U would be applied only if ϕ holds.
- U1 || U2 (parallel update) Updates U1 and U2 happens at the same time in the same state.
- {U1}U2 (update application) For updates U1 and U2, update U2 is applied in the state U1 describes.

2.4 Symbolic Execution

Executing programs with symbolic values instead of concrete values is called symbolic execution. In fact, a symbolic value substitutes all concrete input values that take the same execution path. Symbolic execution covers all possible paths in a given program to a certain depth. A program may have deep execution paths, if there exists either loops or recursion in it. As a simple example, imagine the following method that does multiplication:

```
int multiply (int x, int y)
{
   int a = x;
   int b = y;

   while  (b > 1) {
     a = a + a;
     b = b - 1;
   }
   return a;
}
```

And for now, just imagine that both x and y are positive values. Now, instead of plugging in a concrete value for x and y, we use symbolic values s0 and s1, and as the program is executed we keep track of symbolic states as well as path conditions:

```
{a = s0} Symbolic State
int b = y;
while (b > 1) {
  a = a + a;
  b = b - 1;
}
return a;
```

,

```
{a = s0, b = s1} Symbolic State
while (b > 1) {
  a = a + a;
  b = b - 1;
}
return a;
```

and now we reach a branch, so we should add it to the path condition:

```
{s1>1} Path Condition
{a = s0, b = s1} Symbolic State
a = a + a;
b = b - 1;
while  (b > 1) {
  a = a + a;
  b = b - 1;
}
return a;
```

,

```
{s1>1} Path Condition
{a = s0, b = s1, a = s0 + s0, b = s1 - 1} Symbolic
  State
while (b > 1) {
  a = a + a;
  b = b - 1;
}
return a;
```

and we again come across the while loop, and assuming, for example, that the guard does not hold this time we have:

```
{s1>1 && s1<=1} Path Condition
{a = s0, b = s1, a = s0 + s0, b = s1 - 1} Symbolic
State
return a;
```

A very first observable thing here is that path conditions get accumulated as symbolic execution goes on. Thus, in the example above, the guard did hold the first time, but not the second time. And, both of them got added to the path condition. By a given path condition a specific path in the program is recognized, while by symbolic state a trace of updates (i.e. any changes in the state during symbolic execution) that occur along the path is kept. What we did above with the while loop is called loop unwinding as we simply did unwind the loop and executed its body. This is not the only way to treat loops in KeY since loop unwinding can potentially last forever. Another alternative is to utilize loop invariant rule which we will discuss in chapter 5.

Another thing to observe is that symbolic execution covers all possible paths to a certain depth. In the example above, we covered all paths with depth one, since we only unwinded the while loop once. Thus, in programs with loops or recursion we can not cover all possible paths of the program, since with any new loop unwinding or recursive call a new condition gets added to the path condition, so a new distinctive path in the program is recognized.

And finally, there might be some paths in a program that never get executed. Such paths are called unfeasible paths. In the above example, imagine if b was a negative number, then any path satisfying the loop guard and executing its body would have been an unfeasible path.

2.5 Design By Contract

Design by contract (DbC) or programming by contract refers to a design approach in software development in which software developers are required to define formal and verifiable interface specifications for software components using pre-conditions, post-conditions and invariants. In fact, DbC is a metaphor on how elements of a software system collaborate with each other, on the basis of mutual obligations and benefits. The supplier must provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit) [6].

- The supplier must provide a certain product (obligation) and is entitled to expect that the client has paid its fee (benefit)
- The client must pay the fee (obligation) and is entitled to get the product (benefit)
- Both parties must satisfy certain obligations, such as laws and regulations, applying to all contracts

The ultimate goal is to prevent bugs based on such a mechanism. The term DbC was used by Bertrand Mayer [9] in connection with Eiffel[5] programming language for the first time.

2.6 Formal Specification

Formal specification means to describe a software or hardware system mathematically. It expresses what the system is intended to do, not how to do it. In software systems, specifications may be used at different phases of development (e.g. requirements analysis, design, implementation etc). But here, we solely deal with specifications at the implementation level. We write specifications in formal languages for Java programs. There are three basic concepts in writing specifications:

i Notion of state: a snapshot of system including all variable values and configurations

ii Notion of transition: a pre-state that gets transited to a post-state by an operation

iii Formal language: a formal specification language to formulate specifications in

2.6.1 Object Constraint Language

KeY supports Object Constraint Language, OCL for short, as a specification languages. OCL is part of OMG standard for UML. Although OCL is a formal language, it emphasizes the fact that people with not so strong mathematical background be able to work with it. To exemplify OCL, let us take the example of a dummy multiplication method:

[5] http://www.eiffel.com

```
int multiply (int x, int y)
{
  int a = x;
  int b = y;

  while (b > 1) {
    a = a + a;
    b = b - 1;
  }
  return a;
}
```

We already mentioned an assumption that both x and y are positive. But, it is indeed possible to formulate it as a pre-condition meaning that in the pre-state of this multiplication operation, x and y should should contain positive values. Also, we expect that at the end of this operation the result of x times y is returned. Suppose that this method is implemented in a class called Math. Then, the specification for this method in OCL would be as simple as something like:

```
context Math::multiplication (x: Integer, y: Integer)
pre: x > 0 and y > 0
post: result = x*y
```

It requires that both x and y are greater than 0 at pre-state and at the post-state it returns x*y as the result, all in the context of the class "Math" and the method "multiplication".

Since, we will concentrate on JML more, we do not discuss OCL anymore here. However, further information and details about OCL can be found in [1]

2.6.2 Java Modeling Language

Another specification language that KeY supports is Java Modeling Language or just JML. JML is a popular specification language for Java programs. There are two main differences between OCL and JML. Firstly, JML is not standardized like OCL and is being developed and maintained by a research community. Secondly, JML directly talks about Java source code meaning that unlike OCL it can not be used to write specifications in software development phases that do not deal with source code.

JML is in fact something more than Java meaning that it is built on top of Java. Thus, there is a close integration between the two and it is possible to use Java expression in JML. Let's again take the simple multiplication example and write a specification for it in JML. Assume we have a class called Math as follows:

```
Class Math {
  ...
  ...
  ...
  private int [] arr;
  public Math(int size){
    arr = new int[]
  }
  ...
  ...
  ...
  /*@ public normal_behavior
  requires x > 0 && y > 0;
  ensures \result == x * y;
  assignable \nothing
  @*/
  int multiply (int x, int y)
  {

    int a = x;
    int b = y;

    while (b > 1) {
      a = a + a;
      b = b - 1;
    }
    return a;
  }
  ...
  ...
  ...
}
```

As it can be seen, we use visibility modifiers for each of the operation contracts. They are useful, for example, when we want to have public access

inside JML specifications to a Java private class member. "normal_behavior" states that this operation contract states a normal behavior (i.e. no exception is thrown) of the method provided that the pre-conditions are met formulated in "requires" part. "ensures" states the post-condition and the assignable clause lists all the variables that are allowed to be changed by this method, which in this case there is none of them. We can add more operation contracts by using "also" between contracts. Moreover, we can define contracts for exceptions that might be thrown by using "exceptional_behavior". Then by using "signals" keyword we define the type of exception. For example we can add another operation contract for the multiply method and write both of the contracts as follow:

```
/*@ public normal_behavior
requires x > 0 && y > 0;
ensures \result == x * y;
assignable nothing
@*/
also
/*@ public exceptional_behavior
requires x <= 0 || y <= 0;
signals (MathException);
@*/
```

The post-state of exceptional_behavior says that a MathException is thrown.

Quantification is supported in JML. There are two types of quantifiers: universal and existential. we use "forall" keyword for universal quantification and "exists" for existential quantification and they are expressed in the following general forms:

```
(\forall t x; a)//for all x of type t, a holds
(\exists t x; a)//there exists x of type t
  fulfilling a
(\forall t x; a; b)//for all x of type t fulfilling
  a,  b holds
(\exists t x; a; b)//there exists a x of type t
  fulfilling a, such that b holds
```

Now, assume we want to state in JML that arr in Math class can not hold negative values. We do so by:

```
(\forall int i; i>=0 && i<arr.length; arr[i]>=0)
```

arr.length refers to the size of arr whatever it is. We could express the

same thing by using an existential quantifier too:

```
!(\exists int i; i>=0 && i<arr.length; arr[i]<=0)
```

Just note that arr is declared as private. In fact, to be able to access arr from anywhere in specifications we should declare it as "spec_public". So, it would be:

```
private /*@ spec_public @*/ int[] arr;
```

To declare a method side-effect free we add a "pure" to it so that if the method changes anything from the state JML would show an error. For example, since the multiply method does not have any side-effects we can declare it as:

```
int /*@ pure @*/ multiply (int x, int y)
```

Class Invariants

A class invariant is an invariant across operations of a class. An invariant is a logical predicate that evaluates to the same value before and after an operation or a sequence of operations. In other words, a class invariant is established once an object from a specific class is constructed (i.e. when the constructor is called) and it constantly holds between calls to public methods, thereafter. We define class invariants in JML by using "invariant" keyword. For instance, assume we want to have a class invariant for the dummy Math class stating that arr can not be null. We do so by:

```
/*@ invariant arr!=null @*/
```

More information and details about JML in [10]

Chapter 3

Visual Symbolic Execution Debugger

This chapter discusses the Visual Symbolic Execution Debugger (VSED) [3], which this work is mainly based upon. A somewhat close look at its different features and capabilities is also included.

3.1 What VSED is

VSED is an omniscient debugger based on symbolic execution and is realized as an Eclipse plug-in. Omniscient debugging is a debugging method in which it is possible to go backward in time. In fact, going backward in time becomes possible whenever we keep record of everything that happens during the execution of a program. Thus, by having the history recorded it is possible to jump back and forth in a program. Since, VSED is based on symbolic execution and in symbolic execution all variables have symbolic values and all the execution history exists in the execution tree, it is not difficult to jump back and forth. Beside this, using symbolic values means that it is possible to start debugging at any given position in the program. Thus, it is not necessary to do complex initializations and certain establishments from where it is possible to start debugging [2]. One more advantage of symbolic debugging approach is its memory efficiency. Since, as mentioned, jumping back and forth in a program run needs a complete record of history, it is deemed to consume a lot of memory, but it is not quite the case in symbolic execution. Since debugging can be started at any position in the program, the length of the execution tree, which holds a complete history of a symbolic run, can be much shorter, hence resulting in less memory

consumption.

VSED runs in Eclipse and users interact with it via Eclipse IDE, but it in fact runs the KeY prover in the background which is the main symbolic execution engine. VSED hands over a given program to KeY prover, and Key prover runs the program symbolically. VSED then extracts the execution tree from the symbolically-executed program and draws it on a view inside Eclipse IDE. VSED also offers debugging functionalities on this tree.

3.2 Execution Tree

Execution tree is a tree structure of a symbolically executed program which includes symbolic states of the program in addition to some other events like method invocation or return from a method call. Execution trees are extracted and constructed from proof trees. Proof trees are constructed by running the automated proof search strategy of KeY. This strategy aims to close the proof tree, hence completing the proof, for a given JML annotated Java file which VSED has handed over to KeY prover. But, it usually can not close the proof tree and needs interaction with user. It happens, for example, because there are more than one rule that can be applied on a branch or because there is a loop, for which in this case a loop invariant has to be provided in order to close all of associated branches in proof tree.

3.3 Execution Tree View

VSED, after constructing the execution tree from proof tree, draws it on a view named Execution Tree View inside Eclipse. Fig 4.1 shows a partial execution tree for a quicksort program (section 4.1). The starting node is always labeled with "Start". Green circles show the end of a branch. Whenever there are branch nodes, the arches coming out of them are labeled with "BC". Method calls as well as method returns are shown as white nodes. "self" in method call nodes refers the object this. Also, as can be seen, the loops are simply unwinded and are drawn as executed. To expand a branch, we right-click on the leaf (in fact, any node colored violet can be selected) of that branch, which corresponds to its associated Java statement, that pops up a menu from which "Run", "Step into", or "Step over" can be done. Step over is different from step into in that it skips through all method calls, if there is any. Example is shown in Fig 3.2.

Also, when a node is selected, the path condition leading to that branch and node can be seen in the Symbolic Execution Debugger View. There

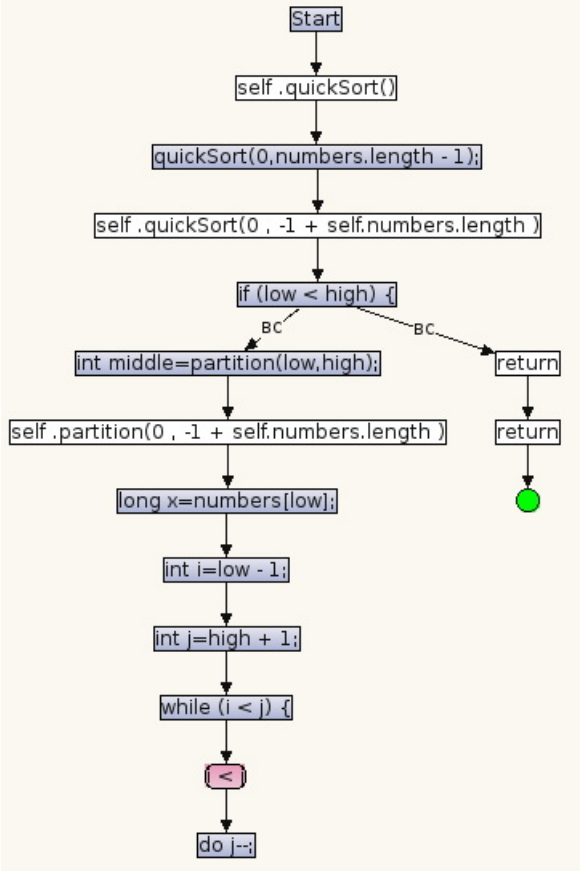

Figure 3.1: Execution Tree View-Example

is also a compartment on the right side of the view that shows the branch condition of currently-selected node. We can accommodate more portion of a large execution tree into screen by adjusting major and minor spacing. Moreover, a double click on a node in the execution tree highlights the associated Java statement in editor.

3.4 Statement Breakpoints

A breakpoint is a stopping place in the program where debugger temporarily suspends execution and enters break mode. In break mode, it is possible to examine variable values of the state and possibly change them too. There are different types of breakpoints (e.g. conditional, line, statement etc). State-

ment breakpoints, as the name reveals, are based on program statements, so breakpoints can be set at statement level, which makes them quite different from line breakpoints in which breakpoints can be set at lines of source code. VSED supports statement breakpoints at the moment. Statement breakpoints in VSED are added or removed in Statement Breakpoint View by pushing add or remove button while the cursor points to a Java statement in the editor. These breakpoints are also useful to prevent expansion of some parts of execution tree that are not of interest. So, we can set breakpoints on those places and be sure that they will no more be expanded, especially when a execution tree tends to grow. Fig 3.2 and 3.3 show an example:

There is also a view called Watchpoints in which it is possible to define watchpoints (a.k.a data breakpoint) by defining so-called watch expressions (Java expressions) such that whenever they become true, the program suspends the execution.

3.5 Symbolic State Visualization

There is another view called Symbolic State View in VSED. It shows symbolic state diagrams. A symbolic state diagram is based on symbolic heaps and symbolic states concept. A symbolic heap is formed from path conditions and updates. It defines a set of states and is given by a first order formula called heap constraint and an update called heap update. The heap constraint imposes restrictions on the possible states and the update describes state changes to these states [3]. Visualizing symbolic heaps is useful to understand the program behavior and realizing what has happened during a symbolic execution. It also helps to find bugs since it shows which inputs lead to which particular symbolic state. A symbolic diagram is a diagram that visualize a symbolic heap. Symbolic diagrams are similar to UML (Unified Modeling Language)[1] object diagrams. An object diagram is a static structure diagram and it focuses on the static structure of the system being modeled, irrespective of time. Basic elements of an Object diagram are objects (instances of classes) and associations (the relationship between instances) between them. Symbolic diagrams also consists of so-called symbolic objects (symbolic instances of classes) and association (directed binary association) between them. An example is given in Fig 3.4 and 3.5, which shows two symbolic state diagrams of a remove method of a linked-list; before (pre-state) and after(post-state) removal of a given element [2]:

[1] http://www.uml.org

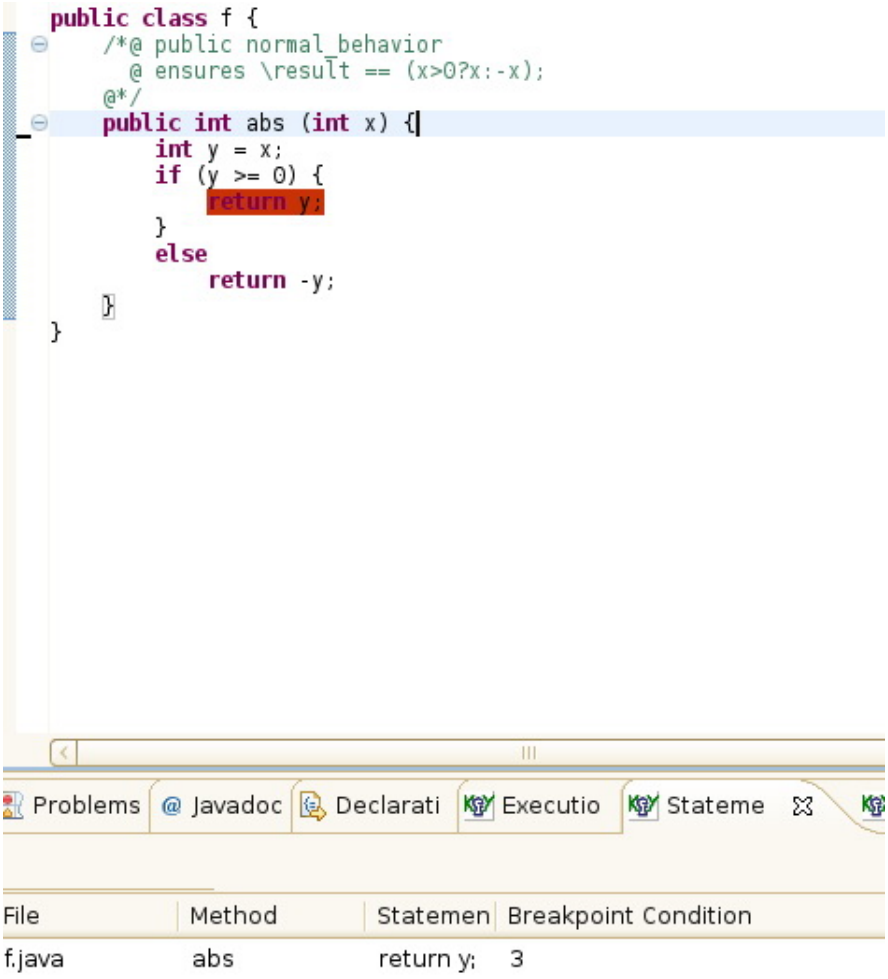

Figure 3.2: "abs" method with a breakpoint on "return y" statement

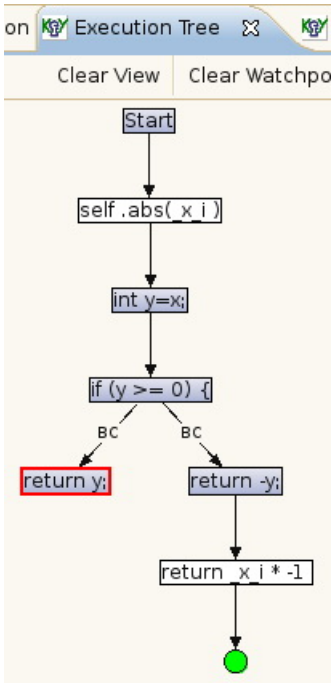

Figure 3.3: Execution Tree View for abs method with a breakpoint on "return y" statement

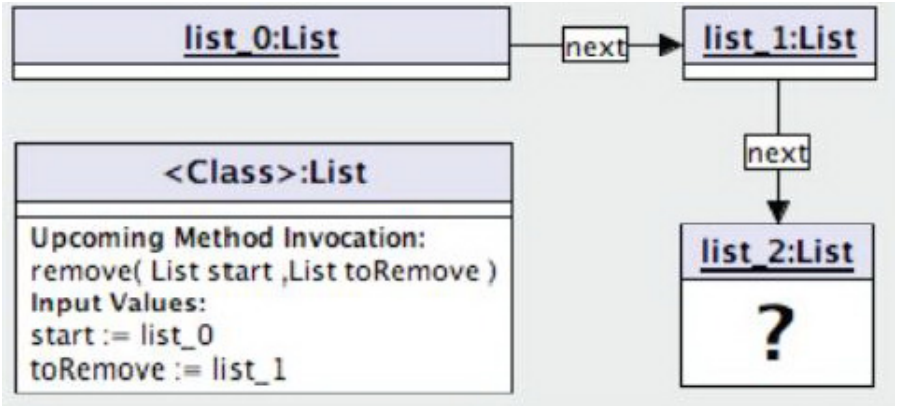

Figure 3.4: Symbolic state before removal of list element list_1 (pre-state)

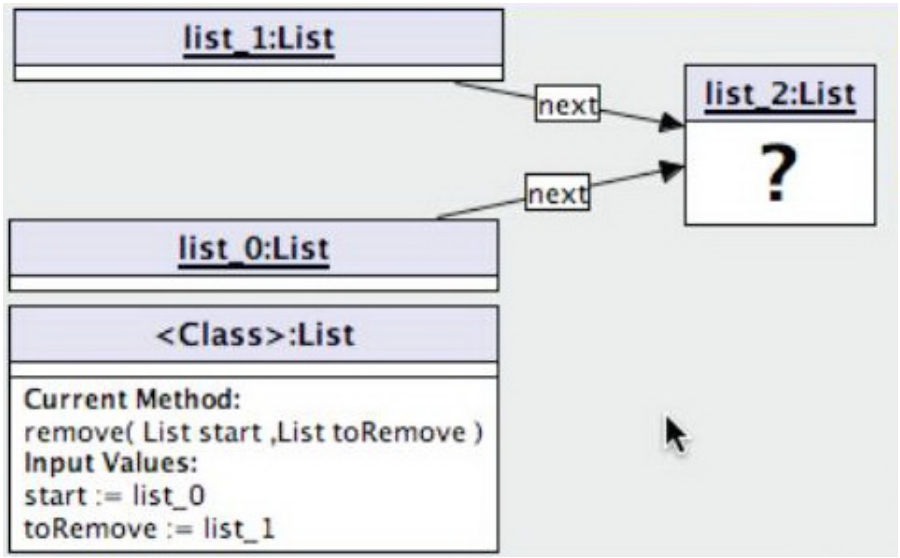

Figure 3.5: Symbolic state after removal of list element list_1; list_1 is no longer reachable from the list head list_0 (post-state)

Chapter 4

Short Evaluation Of VSED

This chapter shortly studies how the Visual Symbolic Execution Debugger (VSED) can possibly work on real programs. For this, we intentionally introduce a bug into the programs and then we try to find the source of infection using VSED. The two examples are a sample implementation of quicksort and mergesort algorithms.

4.1 Quicksort

Below is an implementation of the famous quicksort algorithm written as a Java class annotated with formal specifications (written in JML) of public methods:

```
public class QuickSort {
  private /*@spec_public@*/ int numbers[];
  public QuickSort( int size )
  {
    if (size < 1) size = 1;
    numbers = new long[size];
    for (int i = 0; i < size; i++)
    {
      numbers[i] = (int)(Math.random() * 100);
    }
  }
  /*@ public normal_behavior
  @ requires numbers != null && numbers.length > 0;
  @ ensures \old(numbers.length) == numbers.length;
  @ ensures (\forall int i; 0 <= i && i < \old(
```

```
  numbers.length);
(\exists int j; 0 <= j && j < numbers.length;
  numbers[j] == \old(numbers[i])));
@ ensures (\forall int i; 0 <= i && i <
numbers.length 1; numbers[i]<=numbers[i+1]);
@*/
public void quickSort()
{
  quickSort(0, numbers.length 1);
}
private /*@spec_public@*/ void quickSort
  (int low, int high)
{
  if (low < high)
  {
    int middle = partition(low, high);
    quickSort(low, middle);
    quickSort(middle + 1, high);
  }
}
private /*@spec_public@*/ int partition
  (int low, int high)
{
  int x = numbers[low];
  int i = low 1;
  int j = high + 1;
  while (i < j)
  {
    do
    {
      j;
    }
    while (numbers[j] > x);
    do
    {
      i++;
    }
    while (numbers[i] < x);
    if (i < j)
    {
```

```
        int tmp = numbers[i];
        numbers[i] = numbers[j];
        numbers[j] = tmp;
      }
      else
        break;
    }
    return j;
    }
}
```

Now, we intentionally change the following:

```
private /*@spec_public@*/ void quickSort
(int low, int high) {
  if (low < high)
  { ...
```

to:

```
if (low <= high)
  { ...
```

So, what actually happens is that recursive calls to the quickSort method will never end. But, we do not know yet that it is a bug. The only thing we get when we run the program is a "java.lang.StackOverflowError". The infection source can be found in conventional debuggers by tracing back the recursive calls in the runtime stack, while having an eye on the 'high' and 'low' variables, where we can find out that low=high=0, which gives a pointer to the source of bug.

However, with VSED, it is possible to catch the bug right away. If we run this program symbolically, we will see that there is only one branch coming out of 'if (low <= high)' node in the Execution Tree (Fig 4.1), while logically there should be another branch which returns (Fig 4.2). It shows that recursive calls never stop, since the condition is always satisfied. It is due to the require part of quicksort() specification, which demands numbers.length > 0. This example shows benefit of a thorough specification as well. Even if there were no such a requirement in the specification, it was still possible to catch the bug by checking the Branch Condition compartment located on the right side of Execution Tree View, which shows numbers.length <= 0. However, it might have required a bit of more attention.

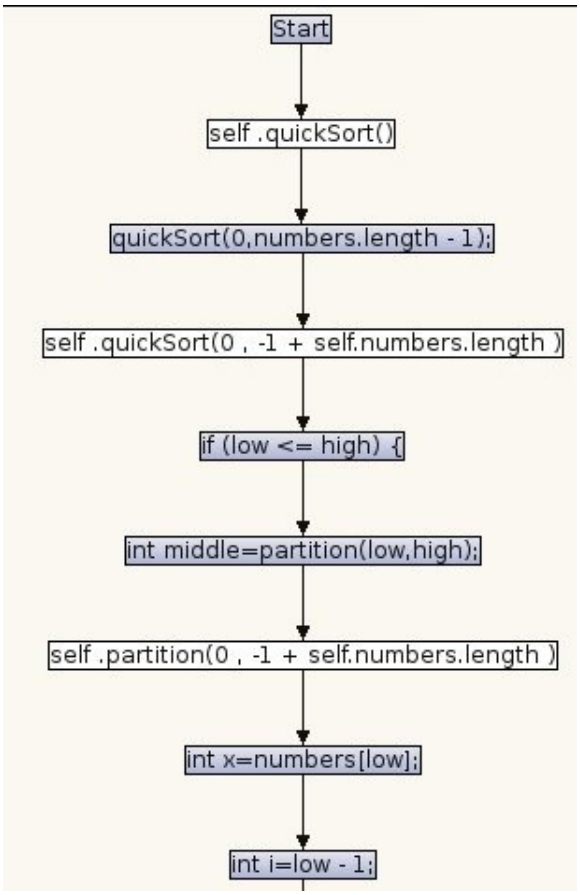

Figure 4.1: Partial execution tree when if (low <= high)"

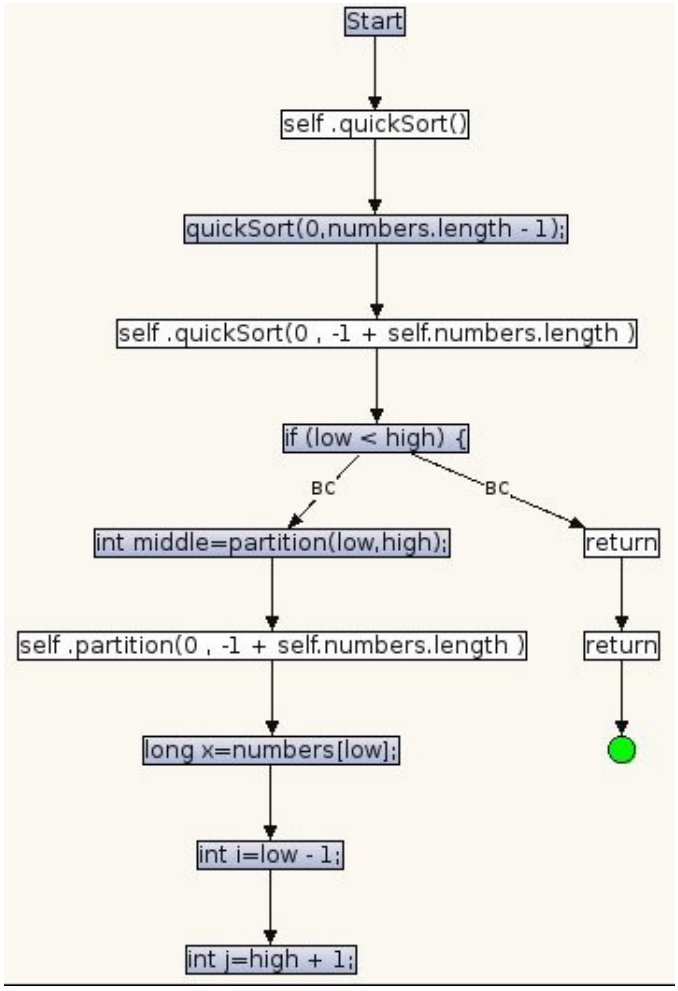

Figure 4.2: Partial execution tree when if (low $<$ high)"

4.2 Mergesort

In this second section we investigate Mergesort algorithm. Again, we try to find an intentionally introduced bug utilizing VSED. Here is the source code before introducing any bugs along with JML specifications:

```
public class MergeSort {
  private /*@spec_public@*/ long[] numbers;
  public MergeSort(int max) {
    if (max < 1) max = 1;
    numbers = new long[max];
    for (int i = 0; i < max; i++)
    {
      numbers[i] = (long)(Math.random() * 100);
    }
  }
  /*@ public normal_behavior
  @ requires numbers != null && numbers.length > 0;
  @ ensures \old(numbers.length) == numbers.length;
  @ ensures (\forall int i; 0 <= i && i < \old(
    numbers.length);
  @ (\exists int j; 0 <= j && j < numbers.length;
    numbers[j] == \old(numbers[i])));
  @ ensures (\forall int i; 0 <= i && i <
    numbers.length-1; numbers[i] <= numbers[i+1]);
  @*/

  public void mergeSort() {
    long[] workSpace = new long[numbers.length];
    recMergeSort(workSpace, 0, numbers.length 1);
  }
  private void recMergeSort(long[] workSpace,
    int lowerBound, int upperBound)
  {
    if (lowerBound == upperBound) // if range is 1,
      return; // no use sorting
    else { // find midpoint
      int mid = (lowerBound + upperBound) / 2;
      // sort low half
      recMergeSort(workSpace, lowerBound, mid);
```

```
    // sort high half
    recMergeSort(workSpace, mid + 1, upperBound);
    // merge them
    merge(workSpace, lowerBound, mid + 1,
    upperBound);
  }
}
/*@ public normal_behavior
@ requires 0 <= lowPtr && lowPtr < highPtr &&
  highPtr <= upperBound && upperBound <
  numbers.length;
@ requires workSpace != null && workSpace.length
  > 0;
@ requires numbers != null && numbers.length
  > 0;
@ requires numbers.length == workSpace.length;
@ ensures \old(workSpace.length) ==
  workSpace.length;
@ ensures \old(numbers.length) ==
  numbers.length;
@ ensures (\forall int i; 0 <= i && i < \old(
  numbers.length);
@ (\exists int j; 0 <= j && j < numbers.length;
  numbers[j] == \old(numbers[i])));
@ ensures (\forall int i; lowPtr <= i && i <
  upperBound; numbers[i] <= numbers[i+1]);
@*/
private void merge(long[] workSpace, int lowPtr,
 int highPtr, int upperBound) {
  int j = 0; // workspace index
  int lowerBound = lowPtr;
  int mid = highPtr - 1;
  int n = upperBound - lowerBound + 1;
  // # of items
  while (lowPtr <= mid && highPtr <= upperBound)
  if (numbers[lowPtr] < numbers[highPtr])
    workSpace[j++] = numbers[lowPtr++];
  else
    workSpace[j++] = numbers[highPtr++];
  while (lowPtr <= mid)
```

```
      workSpace[j++] = numbers[lowPtr++];
    while (highPtr <= upperBound)
      workSpace[j++] = numbers[highPtr++];
    for (j = 0; j < n; j++)
      numbers[lowerBound + j] = workSpace[j];
  }
}
```

Since this time the program is a bit longer and tracing a bug might be more subtle, private void merge(), which is used by mergeSort() is also specified by JML. Although it is a private method and there is usually no need to specify private methods, having them specified gives this possibility to run VSED for them and investigate them independently.

One interesting thing is that VSED does not only help to debug and improve code, but also specification. For example, if we exclude the first require clause from merge()'s specification, we will get the following execution tree, having three branches coming out from "while (lowPtr <= mid && highPtr <= upperBound)" node. But having the first require clause included in the specification, two of those three branches will be eliminated, since the precondition has filtered them. And it really is correct since this method is only called from recMergeSort() and according to that, this while loop would execute at least once under any call.

Now imagine if we had this:

```
int n = upperBound - lowerBound;
```

instead of:

```
int n = upperBound - lowerBound + 1;
```

This is not a complicated bug, but such bugs are usual and sometimes subtle to catch. In this case, it does not cause any failure but causes incorrect outputs. Even worse is that it sometimes alters data, for instance:

Input: 3 7 8 7 5
Output: 3 5 5 7 5

But, it also sometimes works properly, like:

Input: 1 6 6 1 7
Output: 1 1 6 6 7

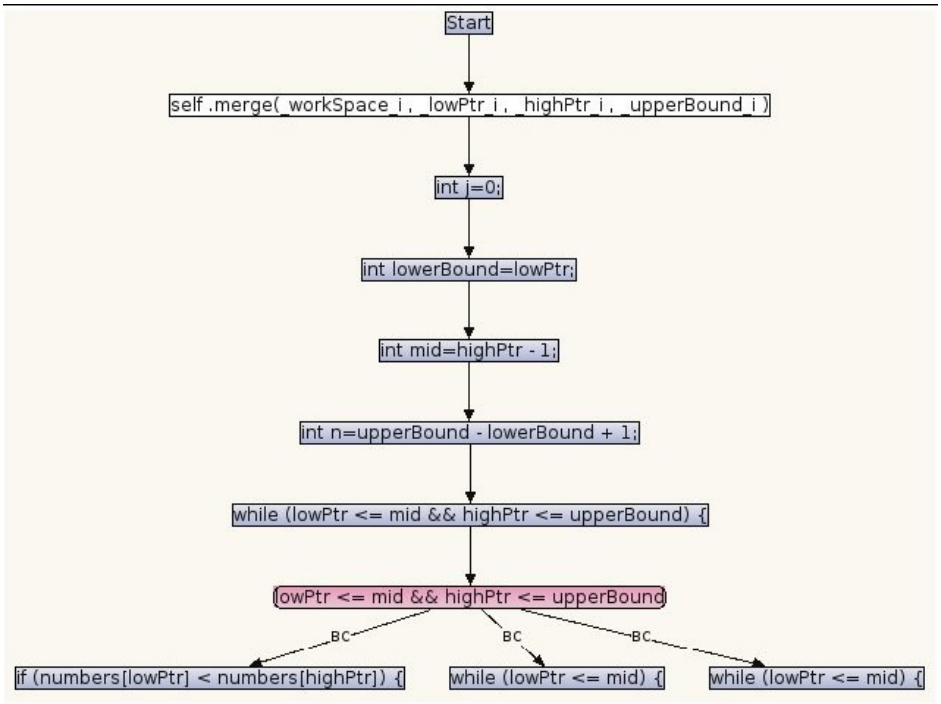

Figure 4.3: Execution tree with a weaker precondition

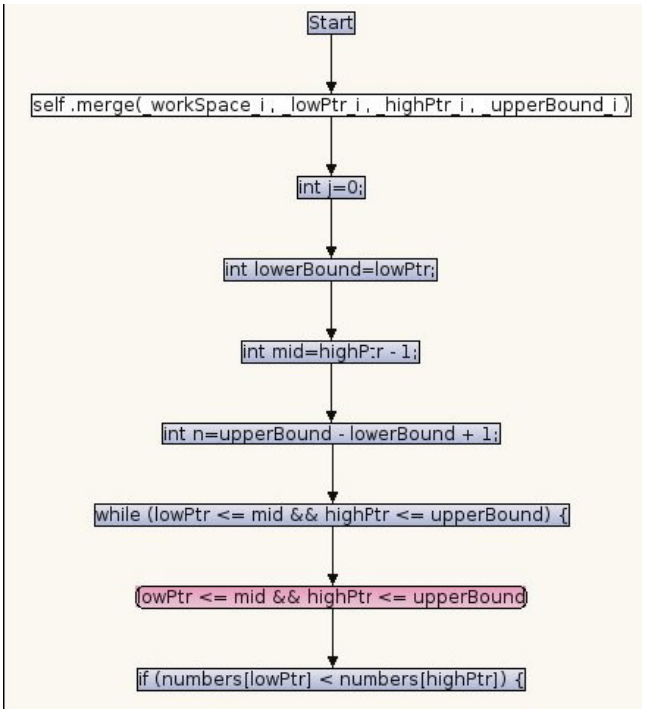

Figure 4.4: Execution tree with a stronger precondition

Catching this bug with conventional debuggers (like eclipse debugger) is possible by toggling breakpoints and investigating variables. Since we do not get any failures or exceptions like previous example and also data become corrupted sometimes, it is improbable that recursive calls are the source of the bug. So, we can conjecture that something is going wrong in merge() method. We can toggle breakpoints at different places through its body and investigate how mid, n, j, lowerBound change and how they affect workSpace and numbers array in different calls. By following up, it becomes clear that the 'for' loop misses to copy one last element each time from workSpace back to numbers array. And since this is 'n' that controls the loop condition and it is data dependent on the line we changed, that line is also not dependent on any other statement, we reach to the source of infection.

However, finding this bug by VSED and only the execution tree is not very straightforward. Since in this case we need direct investigation of the variables, watch points and the visualization facility are more helpful. One way to catch the bug with the help of watch point is to check if workSpace and numbers array elements are equal after each return from merge method for indices from lowerBound to upperBound. By evaluating this watch point for the return node (Fig 4.6), we will get the clue into the bug and we can find the bug by following up control dependency hierarchy as was described above.

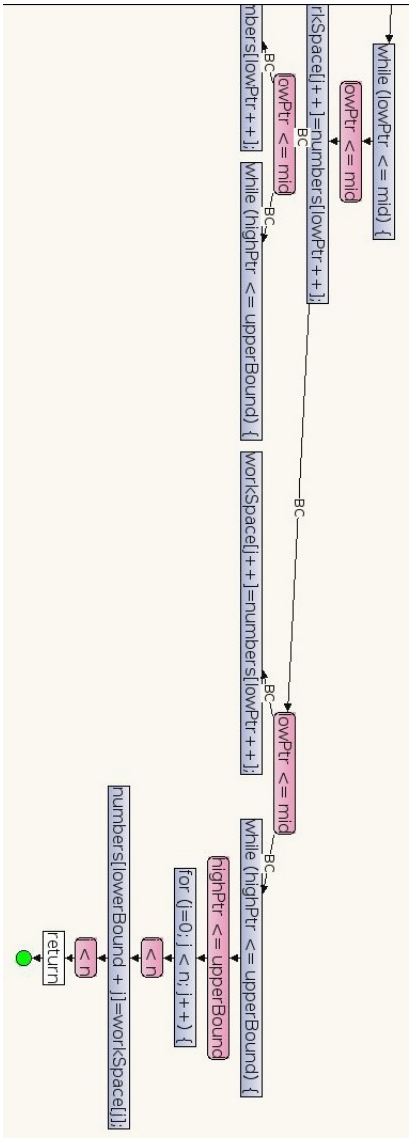

Figure 4.5: Partial execution tree for merge()

Chapter 5

Towards Finite Execution Trees

This chapter discusses loop invariant rule in KeY prover and how to integrate and visualize it in the new verification interface inside Eclipse.

5.1 Loop Invariant

A loop invariant is a statement of conditions (i.e. a logical predicate) that evaluates to true upon entry to a loop as well as during iterations of the loop. It implies that on exit from the loop, the loop invariant and loop termination condition holds. Typically, a loop invariant is denoted by I. Assume the following while loop:

```
while (x<=20) {
  x=x+1;
}
```

I = {x<=20} would be good enough as an invariant for this loop. It holds both on entry as well as during iterations of the loop. We should note that it is always possible to state a weak loop invariant (e.g. I = {true}), but it is not useful, as we will see later, to prove anything with. In KeY, we can provide loop invariants by either writing them as JML comments in Java file or by entering them interactively in a dialog box provided by KeY.

5.2 Loop Invariant Rule

In KeY, we can prove loops by either induction or utilizing the loop invariant rule. we only discuss the first one here. Loop invariant rule is a deductive rule that proves loops correctness given a suitable loop invariant. It looks like as follows (in formula format):

$$\text{LoopInvRule}\ \frac{\Gamma \Longrightarrow U[I], \Delta \quad I, se \Rightarrow [p]I \quad I, !se \Rightarrow \phi}{\Gamma \Longrightarrow U[while(se)p]\phi, \Delta}$$

As we see, applying a loop invariant rule results in three branches that should be proved one by one. The first branch ($\Gamma \Rightarrow U[I], \Delta$) is called "initially valid". As the name reveals, it proves that the invariant holds upon entry into the loop. The second branch ($I, se \Rightarrow [p]I$) called "body preserves" proves that the invariant indeed holds during iterations of the loop. And the last branch ($I, !se \Rightarrow \phi$) called "use case" proves that the invariant holds, whereas the loop condition does not. Note that this rule proves partial correctness of a loop. To prove total correctness, loop termination should be proved too (it is done by updating the above-mentioned rule by adding a so-called loop variant which is an integer term, like v, to the rule and prove that it varies in an interval of 0 to some integer bound like v0 during its iterations).

5.3 Loop Invariant Rule Visualization

As mentioned earlier, to be able to fully verify programs we need to have finite execution trees, and for that, we need to have support for rule invariant rule. VSED unwinds the loop, but one of the goals of this work has been support for loop invariant rule inside the new GUI. To achieve that, we have tried to provide an interactive visualization of loop invariant rule in Execution Tree View such that whenever a loop is met during symbolic execution, user is given with two options of either unwinding the loop or applying loop invariant rule.

When a loop invariant rule is applied, three branches (initially valid, body preserves, and use case) are produced among which initially valid and use case are done on a pure logical level. So, they are not part of symbolic execution, hence execution tree. And in fact, right now there is no way to interact with these two branches via Execution Tree View, but the point is that if the provided loop invariant is good enough, these branches usually get closed automatically without any need of user interaction. On the other

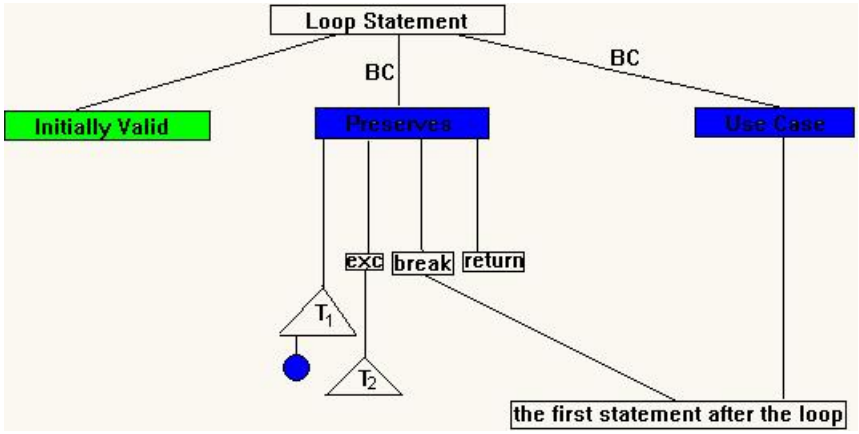

Figure 5.1: Loop Invariant Rule Visualization (before proof)

hand, body preserves executes code (i.e. loop body) and it adds nodes to the execution tree, so those nodes get added under the body preserves branch.

It should also be noted that an abrupt termination might occur within loop body. It can happen if there is an exception thrown, or there are either return or break statements. If it is either an exception or break statement, the program continues to the first statement after the loop, whereas if it is a return statement, no more statements are executed afterwards.

Based on what was mentioned above, a natural way to visualize loop invariant rule would be something similar to Fig 5.1 and 5.2:

If the loop is initially valid and the branch is closed in proof tree, "Initially Valid" branch would be green. In "Preserves" branch, the loop body gets executed. A few things may happen here. If there is any exception thrown in the loop body, then the loop is not executed normally anymore and the thrown exception will either be caught by a catch statement or remain uncaught. In Fig 5.1 and Fig 5.2, T2 refers to execution tree of associated catch statement, if any. If there is any break statement in the loop body, then the loop breaks and does not execute anymore, and the program continues on the first statement after the loop. It might also be the case that there is a return statement withing loop body. In this case, there is no execution of the current method. Finally, if non of these happens, loop continues its iterations normally until finished. T1 refers to execution tree associated with loop body in Fig 5.1. and 5.2.

From the users' view, when execution tree reaches a loop, it stops to give the user possibility of choosing between loop unwinding or applying

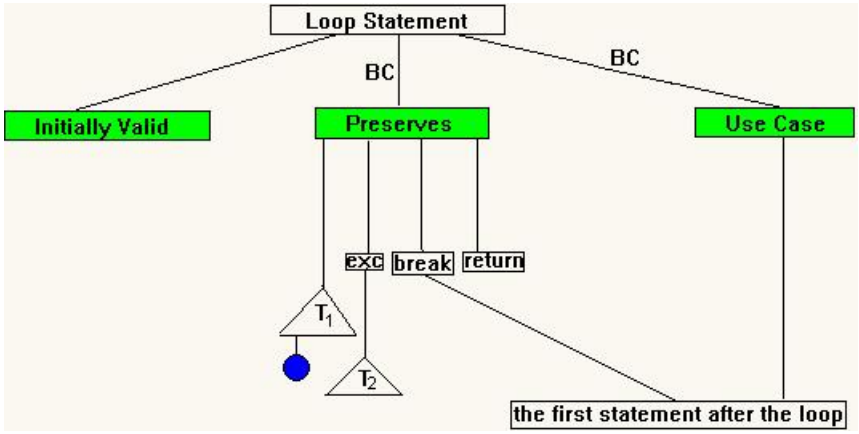

Figure 5.2: Loop Invariant Rule Visualization (after proof)

loop invariant rule. If the loop invariant is provided as JML annotations in the Java file, it is applied automatically. If not, a dialog box comes up and asks for a loop invariant, which is yet to be implemented.

Chapter 6

Modular and Project-Based Verification

This chapter discusses two things: project-based verification that relates to adjusting KeY settings directly from Eclipse, and modular verification which is about method contract rule.

6.1 Project-based Verification

6.1.1 Eclipse IDE

This section gives a general brief overview of Eclipse Integrated Development Environment (IDE).

Eclipse platform Architecture

Eclipse is a project launched by IBM in 2001. It is an open-source framework that is written in Java and is mainly used as a popular IDE. Eclipse, at its core, consists of a small piece of runtime engine called "Platform Runtime" that is responsible to start the platform and run all the other so-called plug-ins. Besides this core, everything else in Eclipse is a plug-in. A plug-in is a piece of code and data that makes a functionality contribution to a system. Thus, all functionalities in the Eclipse IDE are from the plug-ins that are organized into a number of sub-systems and are run on top of the platform runtime. These plug-ins can define so-called "Extension Points" where other plug-ins can add functionality to this plug-in thorough these places. A minimal set of plug-ins required for building a rich client application is called Rich Client Platform (RPC). Thus, Eclipse IDE itself

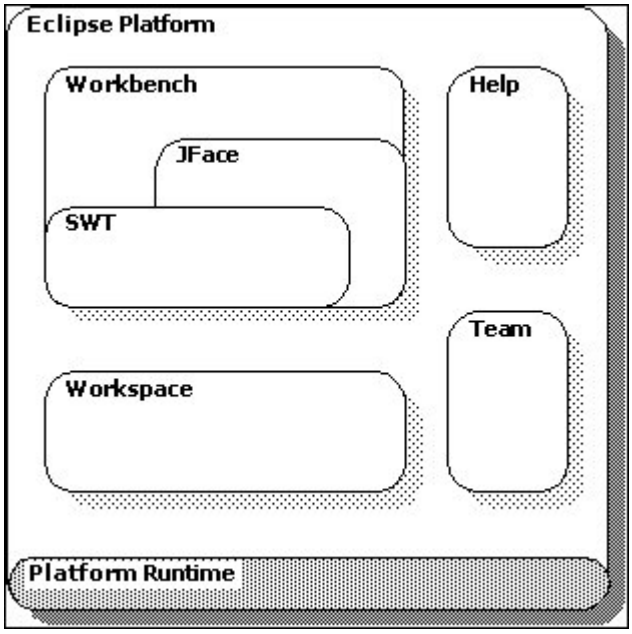

Figure 6.1: Eclipse Platform Architecture

is a RPC application. Fig 3.1 depicts a general overview of Eclipse Platform Architecture.

Workbench

Workbench is one of the subsystems on top of the platform runtime that refers to desktop development environment and utilizes two plug-ins, Standard Widget Tool (SWT) and JFace. SWT is a low-level widget toolkit for Java which provides access to the UI facilities of operating system. JFace is a higher-level application built upon SWT and provides for example dialogs, actions, etc.

The workbench aims to achieve seamless tool integration and controlled openness by providing a common paradigm for the creation, management, and navigation of workspace resources [5]. Each workbench consists of one or more so-called perspectives. Perspectives define the initial set and layout of views in the Workbench window. Each perspective contains views and editors and aims to accomplish a specific type of task (e.g. Java perspective aimed for editing Java source code, Debug perspective aimed for debugging Java programs, etc). Views provide an alternative presentation way to nav-

igate resources and information inside workbench (e.g. Project Explorer view presenting projects and their associated resources). Editors are areas to edit a file resource.

Eclipse SDK contains two plug-ins: Java Development Tool (JDT) and Plug-in Development Environment (PDE). JDT implements a Java IDE and provides tools to develop Java applications. PDE provides an environment and tools to develop plug-ins and extensions.

6.1.2 KeY Project

Whenever we initiate visual debugging or verification through Eclipse, KeY is loaded in the background and will be running thereafter to perform symbolic execution and formal proofs. In KeY, the primary modules that we verify are methods that are members of classes. However, in Eclipse we deal with Java projects that include a number of classes and each class also includes a number of methods.

Once we start to verify Java projects in Eclipse, one desirable thing to have is a way to store KeY settings for each project so that whenever the project is being debugged/verified, these settings are loaded and applied in KeY by default.

To achieve this, a new type of project called KeY-project was added to Eclipse projects. KeY project is a typical Eclipse Java project plus KeY settings. Therefore, a new extra page was added to the Eclipse Java wizard for adjusting KeY settings, which is shown in Fig 6.2:

As it can be seen in Fig 6.2, it is possible to set the taclet libraries that are to be loaded. Taclet libraries add more rules to the rule base. Decision procedures settings are included too. The default decision procedure is Simplify. It is also possible to set the specification language, which is naturally set to JML by default. Finally, the policy of handling nulls as well as integer treatment can be adjusted. Null handling policy specifies if nullpointer checks should be performed when evaluating reference access expressions. If turned off, no "NullPointerExceptions" will be raised when dereferencing a null reference. Integers can also be treated in following three ways [7]:

- javaSemantics (Java semantics): Corresponds exactly to the semantics defined in the Java language specification. In particular this means, that arithmetical operations may cause over-/underflow. This setting provides correctness but allows over-/underflows causing unwanted side-effects.

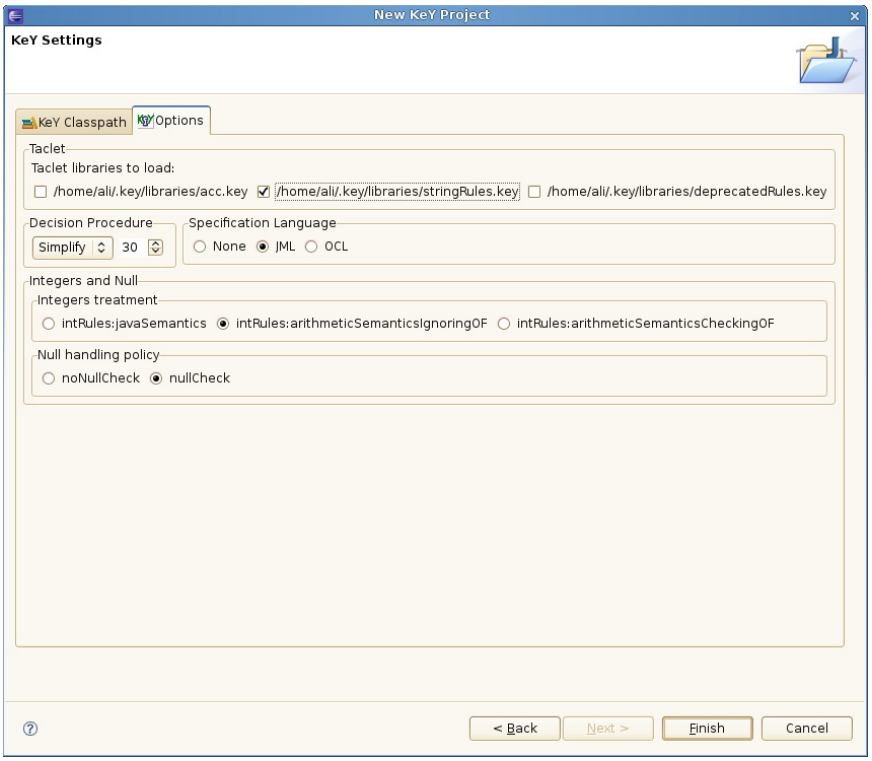

Figure 6.2: An extra page added to Java wizard containing KeY settings

- arithmeticSemanticIgnoringOF (Arithmetic semantics ignoring overflow, default): Treats the primitive finite Java types as if they had the same semantics as mathematical integers with infinite range. Thus this setting does not fulfill the correctness criteria.

- arithmeticSemanticsCheckingOF (Arithmetic semantics prohibiting overflow): Same as above but the result of arithmetical operations is not allowed to exceed the range of the Java type as defined in the language specification. This setting not only enforces the Java semantics but also ascertains that no overflow occurs.

Implementation

Different project types in Eclipse are in fact of different so-called natures. A nature in Eclipse realm means associating life cycle behavior with a project. In simpler words, project natures allow a plug-in to tag a project as a specific kind of project[5]. A project can have one or more natures. New natures can be defined using "org.eclipse.core.resources.natures" extension point. There are two rudimentary terms that plug-in development in Eclipse revolves around: Extensions and Extension points. Every Eclipse plug-in can define so-called extension points by which it gives a possibility to other plug-ins to add a new behavior to this plug-in. On the other hand, an extension in a plug-in provides the interface for other plug-ins to use the plug-in's functionalities.

KeY project has a nature of typical Java projects plus that it extends their behavior to accommodate what is needed to store KeY settings too.

6.2 Modular Verification

Modular verification refers to the idea of focusing on verifying one module at a time. We do verification of any method based on its operation contract(s), but there can be method calls inside it too. Whenever there is a method call, we have two choices of either in-lining the method body or applying the method effect using its specifications. The latter one has a number of advantages. First, it results in a smaller execution tree in the view, so it is easier to handle and deal with the whole execution tree. In fact, this is an important observation since exploiting the specifications is indispensable in order for program verification to scale up. This way, each method only needs to be verified (i.e., executed symbolically) once. In contrast, in-lined methods may have to be symbolically executed multiple times, and the size

of the proofs would grow more than linearly in the size of the program code to be executed symbolically. Moreover, the source code of a (library) method may not be available. Then, the only way to deal with the invocation of the method is to use its specifications [4]. In addition to these, when we use an operation contract, the current state is required to satisfy the pre-conditions. If pre-conditions are not satisfied, it points it out to us that this is not a right place to make the method call. It can be helpful in debugging too, since making a misplaced method call might be very well an infection source.

An operation contract OCm = (Pre; Post; mod; term) for a method m consists of[8]:

- the precondition formula Pre
- the postcondition formula Post
- a set of locations mod ("modifies/assignable clause")
- a termination marker term $\in$ {partial, total}

It means that method m fulfills its contract OCm = (Pre; Post; mod; term) if:

- when it is called the pre-conditions stated as "Pre" hold,
- m terminates/if m terminates, then the postconditions stated as "Post" hold,
- locations stated in mod may have been changed

whether "m terminates" or "if m terminates" depends on the term used. If it refers to the partial term (box modality) it means that there is no guarantee that m terminates, but if m terminates then it the Post must hold. however, if it refers to the total term, then it gives the guarantee on m's termination. The three branches of method contract rule written in formula format are as follows[8]:

$$\frac{\Gamma \Longrightarrow Upre, \Delta}{\Gamma \Longrightarrow U < \pi result = m(a1, ..., an)\rho\omega > \phi, \Delta}$$

$$\frac{\Gamma \Longrightarrow U\nu mod(exc \doteq null \& normalPost \rightarrow < \pi\omega > \phi, \Delta}{\Gamma \Longrightarrow U < \pi result = m(a1, ..., an)\rho\omega > \phi, \Delta}$$

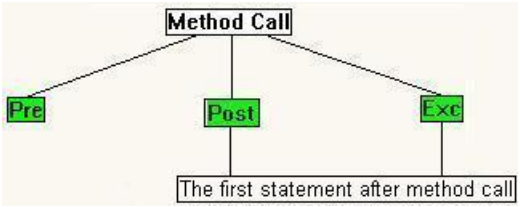

Figure 6.3: Method contract rule visualization

$$\frac{\Gamma \Longrightarrow U\nu mod(exc \doteq null \& normalPost \rightarrow < \pi\omega > \phi, \Delta}{\Gamma \Longrightarrow U < \pi result = m(a1, ..., an)\rho\omega > \phi, \Delta}$$

The first branch checks if the pre-conditions hold. The second and third branches check if the post-conditions hold when there is no exceptions thrown and when there is, respectively. In the conclusion part, the method result gets reflected in the state, if any, by modifying the locations contained in "mod". Also, note that the rule mentioned above have diamond modality meaning that it is meant to prove the total correctness.

For visualization, the three branches in the rule correspond to the three branches shown in Fig 6.3. If the pre-conditions do not hold, then the "Pre" branch can not be closed. In such a case, the corresponding branch color in the tree would be blue. Otherwise, if it can be closed, it would be shown as green. The two other branches would also be green if their corresponding branches in the proof tree are closed (one for exceptional behavior of the method and one for its normal behavior), implying that post-conditions hold. Usually, these branches should get closed automatically. However, if any of these can not be closed, it is not possible to do anything about it right now through Eclipse and the GUI, since it does not operate on pure logical level rules for now.

Chapter 7

Conclusion

This chapter discusses the contributions of this work, some conclusions that can be drawn, and possible routes of future work.

7.1 Contributions

There are three contributions made by this work. They were discussed in chapter 4, 5, and 6. A very short evaluation of VSED is given in Chapter 4. First, intentional bugs were introduced into programs and then it was tried to catch them by VSED. The next thing discussed was support of loop invariant visualization in execution trees. As it was mentioned earlier, to be able to verify programs we need to have finite execution trees and for that, the loop invariant rule has to be applied on loops instead of loop unwinding. A way of visualization for loop invariant rule was suggested in chapter 5. The second thing discussed was project-based verification. In order to save all the KeY settings for each Java project in Eclipse, a new type of project called Key Project was introduced into Eclipse. It was discussed in chapter 6. And, the last discussed thing was support of method contract rule. There are a number of advantages to use method contracts instead of in-lining methods whenever there are method calls inside the method being verified, which were also explained in chapter 6.

7.2 Conclusions

Utilizing formal methods usually needs expert knowledge of the field. One natural step towards making them more accessible and comprehensible is

to find ways that non-experts can use them too. Doing formal verification in KeY requires knowledge and understanding of formal specifications, dynamic logic, symbolic execution, etc. In fact, performing formal proofs in KeY for even simple examples might easily get tricky and tiresome for people with little knowledge or experience. Thus, providing a simpler alternative way of utilizing (basic) KeY functionalities within a prevalent environment(i.e. Eclipse IDE) has seemed a useful step towards making it more accessible and more comprehensible for non-experts (or those who are new to the field).

Since pure logic level rules are pruned away in execution trees drawn in Execution Tree View, it is easier to comprehend them and work with them. In fact, besides the advantages of a symbolic omniscient debugger that were discussed in chapter 4, having a tree-like visualization of programs can be seen as a way to comprehend them better. Having support for loop invariant and method contract rules give the possibility of performing full verification on finite and smaller generated execution trees.

One other observation is that debugging and verification are integrated in one place. So, for example it is possible to start with debugging and once we are confident enough that there are no more bugs, we can go further and try to formally prove the correctness of the program.

7.3 Future Work

Currently, VSED and the visual verification facilities are at prototypical level. A serious testing of the current status of the work, adding appropriate supports for intuitive facilities and functionalities in either debugging or verification part, performing case studies etc definitely help to push the work further into a more full-fledged level. Besides this, there are two main possible routes of future works: debugging and verification. For the debugging, a list of future works is given in [3]. For the verification part which has been the main focus here, some future works are listed below:

i **Finishing the implementation**: A very first step is to finish the implementation part of loop invariant and methods contract rules. The implementation work has been partly done, and the next step is to finish it based on the suggested visualization approaches given in chapter 5 and 6.

ii **Concurrent Programs**: For now, it is not possible to debug or verify concurrent programs. An approach for deductive verification of concur-

rent program has been developed [11]. So, a natural next step would be adapt to the new GUI so that it is possible to debug and verify concurrent programs too.

iii **Counter Example Generation**: Proving JavaCardDL formulas invalid based on generating a counter example is given in [12]. One future work can be to accommodate that.

Bibliography

[1] Object constraint language specification, version 2.0, 2009. www.omg.org/technology/documents/formal/ocl.htm.

[2] Wolfgang Ahrendt, Richard Bubel, and Reiner Hähnle. Integrated and tool-supported teaching of testing, debugging, and verification. In *proceedings of the Second International Conference on Teaching Formal Methods*, 2009.

[3] Marcus Baum. Debugging by visualizing symbolic execution. Master's thesis, Chalmers University of Technology, 2007.

[4] Bernard Beckert, Reiner Hähnle, Peter H.Schmitt, and et al. *Verification of Object-Oriented Software: The KeY Approach.* Springer-Verlag, 2006.

[5] Eclipse Online Documentation, 2009. http://www.eclipse.org/documentation/.

[6] Eiffel Documentation, 2009. http://docs.eiffel.com/.

[7] KeY Quicktour for JML, 2009. http://key-project.org/casestudies.

[8] Reiner Hähnle. Formal analysis of java programs with key, 2009. http://i12www.iti.uni-karlsruhe.de/ key/costws09/JavaDL.pdf.

[9] Bertrand Meyer. Applying design by contract. In *Computer (IEEE), 25, 10*, 1992.

[10] Java Modeling Language Home Page, 2009. http://www.cs.ucf.edu/ leavens/JML/.

[11] Philip Rümer. Sequential, parallel, and quantified updates of first-order structures.

[12] Philipp Rümmer and Muhammad Ali Shah. Proving programs incorrect using a sequent calculus for java dynamic logic. In *Testing and Proofs, Zurich*, 2007.