2022

Translations to Support Loop Invariant Generation in JML

Kohei Koja
University of Central Florida

Translations to support loop invariant generation in JML

by

Kohei Koja
B.E. Computer Science and Engineering, Waseda University, 2020

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2022

Major Professor:
Dr. Gary T. Leavens

ABSTRACT

Software is used in many critical systems in the real world such as autonomous cars and medical devices. Such software must be reliable to protect the general public. One standard way to make reliable software is to use Hoare-style verification techniques. However, for Hoare-style verification of loop correctness, loop invariants are necessary but are difficult for people to write themselves. Since Java is one of the most popular programming languages in the world, it is useful to have a tool to generate loop invariants for Java programs. OpenJML is a widely used program verification tool for Java. However, it does not provide automatic loop invariant generation. Therefore, the problem that this thesis addresses is to automatically generate loop invariants in OpenJML. The plan is to use a third-party tool to achieve this goal. Since this external tool does not support Java, a package is necessary to translate from Java/JML to the syntax of the loop invariant tool. However, there are a few conditions that the package has to meet. Since the programming language that the third-party tool supports has a limited grammar, we have to work around unsupported syntax. The resulting package is successful because it can integrate a loop invariant generation tool with OpenJML. Therefore, this package may help developers produce more reliable Java programs.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1
INTRODUCTION

OpenJML is a program verification tool that can verify Java programs specified in *Java Modeling Language* (JML). Though OpenJML supports various verification features such as a static and run-time checking, it does not support automatic loop invariant generation. Therefore, the problem is to automatically generate loop invariants in OpenJML.

I planned to use an external loop invariant generation tool to integrate that feature in OpenJML. The loop invariant generation tool acts as a black-box that automatically generates a loop invariant. Rapid/Vampire [1, 2] was chosen as such tool. Rapid generates semantics of a program and Vampire generates loop invariants from those program semantics generated from Rapid. However, Rapid takes programs written in its own syntax and specifications written in the SMT-LIB language which is the standard language for inputs and outputs for *Satisfiability Modulo Theories* (SMT) solvers. A SMT solver is a program that can automatically decide if a first-order formula is satisfiable. The goal of this thesis is to develop two translation packages to solve the problem as shown in Fig. 1.1. One is an extension of OpenJML and it translates Java/JML into the syntax of Rapid and SMT-LIB. The other one is an extension of jSMTLIB (available on GitHub [3]) and it translates the SMT-LIB language to JML.

The main contribution of this thesis is designing and building two translation packages between Java and the Rapid's programming language as well as JML and the SMT-LIB language.

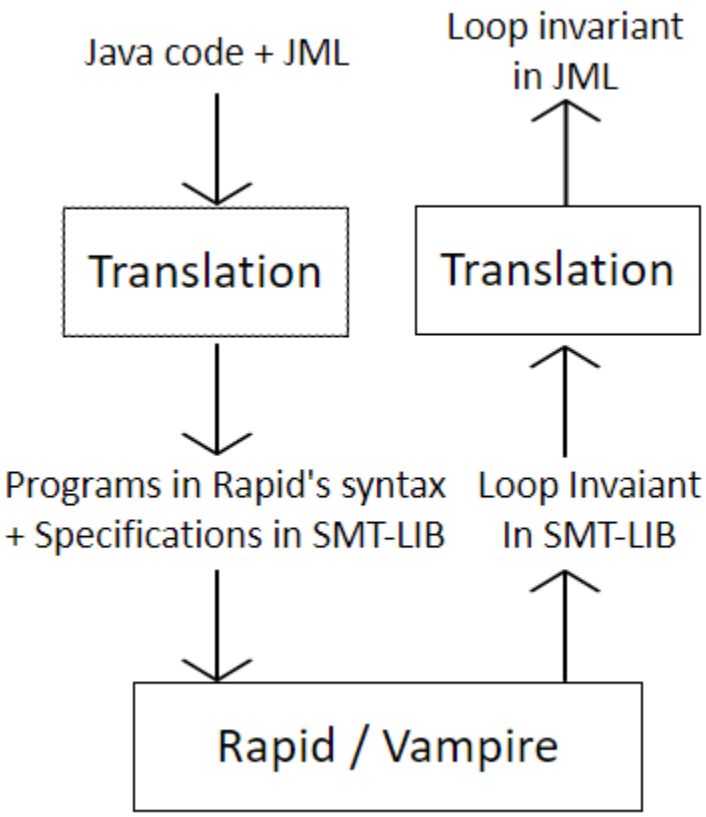

Figure 1.1: Overview of the tool's architecture

CHAPTER 2
BACKGROUND

In this chapter, I will explain some background about program verification.

2.1 Natural deduction rule

Natural deduction is a system to prove if a mathematical proposition is always true or not. It consists of a set of rules of inference to derive conclusions from premises. For example, modus ponens is one of the inference rules which states that if P is true and P implies Q, then Q is true. It is written as follows:

$$\frac{P, P \Rightarrow Q}{Q}$$

Here, the propositions above the horizontal line are the *premises* and the one below the horizontal line is the *conclusion*.

Generally, the natural deduction rule is written as follows:

$$\frac{\alpha, \beta}{\gamma}$$

This means that if the premises of the rule, α and β, both hold, the conclusion γ also holds. In other words, to make γ hold, it is sufficient enough to prove both α and β.

The rule without any premise is called *an axiom* and written as follows:

$$\frac{}{\gamma}$$

These notations of inference rules are used in Hoare Logic which is explained below.

2.2 Hoare Logic

A program is said to be correct when it behaves as its specifications describe. The partial correctness of the program requires the program to be correct under the condition that it terminates normally. On the other hand, the total correctness of the program requires to prove the partial correctness of the program as well as its normal termination [4]. Hoare Logic is a system which uses a set of natural deduction rules for proving program's correctness. A Hoare triple is a feature of Hoare Logic and it describes how the executions of code change the state of the program. Its form is written as follows:

$$\{P\}\ S\ \{Q\}$$

where S denotes a statement, and P and Q denote assertions, called the triple's precondition and postcondition, respectively. This means that if the execution of the statement S starts in a state that satisfies the precondition P, and if S terminates normally, then the final state of its execution satisfies the postcondition Q. Since it assumes termination of S, Hoare logic can only prove the partial correctness of a program [5].

Hoare Logic provides axioms and inference rules of a programming language with use of the Hoare triple. For example, the verification rule for a while loop is written as follows.

$$\frac{\{B \wedge I\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{\neg B \wedge I\}}$$

where I and B are conditions. Here, the condition I must always hold before and after each execution of S in while loop, and it is called *a loop invariant*,. The condition B is a loop condition; it holds before the execution of S but does not hold after exiting the while loop.

The discussion about partial correctness given above did not take termination into account. Termination is an additional proof obligation for total correctness. To prove the total correctness of the program that contains while loops, we need to prove its partial correctness and its normal termination. However, the proof of termination cannot be completely automated because the Halting Problem is undecidable [6]. Since OpenJML's verification uses Hoare Logic, it only tries to verify the partial correctness of Java programs.

If a verifier can find a loop invariant I that satisfies the following three conditions, then it is called *inductive* and could prove the program S is partially correct.

- $P \Rightarrow I$
- $\{I \wedge B\}\ S\ \{I\}$
- $(I \wedge \neg B) \Rightarrow Q$

In other words, finding a correct loop invariant is needed to prove the partial correctness of a program that contains loops.

2.3 Finding a loop invariant

As an example, we consider proving the partial correctness of the following program:

Listing 2.1: An example of a while program [7]

```
1 //@ assume z == 1;
2 while (z * z != 16) {
3     z = z + 1;
4 }
5 //@ assert z == 4;
```

In this example, the loop condition B is $(z * z \neq 16)$, the precondition P is $(z = 1)$, and the postcondition Q is $(z = 4)$.

We consider two loop invariants I such as $(z \geq 1)$ or $(z * z \leq 16)$. Though there are infinite number of invariants, we want the weakest inductive one.[1]

The first loop invariant $(z \geq 1)$ is not inductive because it does not satisfy $(I \wedge \neg B) \Rightarrow Q$ *i.e.,* $(z \geq 1) \wedge \neg(z * z \neq 16) \Rightarrow (z = 4)$.

Similarly, the second loop invariant $(z * z \leq 16)$ is also not inductive because it does not satisfy $(I \wedge \neg B) \Rightarrow Q$ *i.e.,* $(z * z \leq 16) \wedge \neg(z * z \neq 16) \Rightarrow (z = 4)$ where the left hand side expression has two possibilities for z; $z = \pm 4$.

Combining two conditions is inductive since it implies $(z = 4)$ *i.e.* $((z \geq 1) \wedge (z * z \leq 16)) \wedge \neg(z * z \neq 16) \Rightarrow (z = 4)$.

Therefore, the resulting inductive loop invariant of program 2.1 is $(z \geq 1) \wedge (z * z \leq 16)$.

Normally, manually finding a loop invariant takes time and effort. Hopefully, Rapid/-Vampire does this automatically on behalf of the human.

[1] A Boolean formula B is *weaker* than a Boolean formula A if $A \Rightarrow B$ [8].

CHAPTER 3
PROBLEM

Though Rapid/Vampire automatically generates loop invariants, they don't have support for Java programs. Vampire's inputs are written in the same format as Rapid's outputs. However, Rapid's inputs are written in its own syntax and the loop invariants that Vampire generates are written in the SMT-LIB language. Therefore, to use Rapid/Vampire for an automatic loop invariant generation tool, we need to translate Java programs into ones that Rapid accepts as well as translate SMT-LIB into JML.

3.1 Translation from Java/JML programs to Rapid

Rapid is intended to be used with the following steps [9].

1. Write a program in the supported language.

2. Write a safety property of the program in the SMT-LIB language.

3. Pass the file containing both the program and the property to Rapid.

The supported language in step 1 has its own syntax. The listing below is the simple grammar of Rapid programs.

Listing 3.1: The grammar of the programs Rapid accepts [10]

```
program := function

function := func main(){ context }

subprogram := statement | context

statement := atomicStatement

             | if( condition ){ context }

                 else { context }

             | while( condition ){ context

                 }

atomicStatement := skip | ...

context := statement; ... ; statement;
```

The problem is translating the Java programs into the ones that follows this grammar.

For the safety property of the program, I had planned to use the precondition and postcondition of the program. This safety property is written in the SMT-LIB language. Therefore, the second problem is translating the JML precondition and postcondition into the SMT-LIB syntax.

3.2 Translation from SMT-LIB into JML

Resulting loop invariants that Rapid/Vampire generates are written in SMT-LIB language. I cannot directly use the same loop invariants for program verification in OpenJML since OpenJML does not support specifications written in SMT-LIB. Therefore, the problem is

that translating the SMT-LIB language back into the JML syntax. For example, the loop invariant of program 2.1 is written in the SMT-LIB language as follows:

```
(and (>= z 1) (<= (* z z) 16))
```

The equivalent loop invariant is written in JML as follows:

```
//@ maintaining (z >= 1) && (z * z <= 16);
```

The major difference between JML and SMT-LIB is that operators come first in binary expressions in SMT-LIB. Another difference is that `null` is not expressible in SMT-LIB.

CHAPTER 4
APPROACH

In this chapter, I describe my approach to the translation problem stated in chapter 3. First, I describe an approach to the problem of translation from Java/JML programs to Rapid/SMT-LIB. There are two translation problems when using Rapid. One is to translate Java programs into the supported programming language in Rapid. The other is to translate JML into the supported SMT-LIB language to provide a safety property of the program in Rapid. Then, I will explain an approach to the problem of translation from Rapid/Vampire's output, in SMT-LIB, to JML.

4.1 Translation from Java/JML into Rapid

In this section, I consider translating Java/JML programs to programs that Rapid accepts with a running example. The following program is to find a maximum number in an array.

Listing 4.1: A method that returns the maximum value of an array

```
1 class Max {
2     //@ requires x != null && x.length > 0;
3     /*@ ensures (\forall int i;
4       @          i >= 0 && i < x.length;
5       @          x[i] <= \result); */
6     /*@ ensures (\exists int j;
```

```
7         @           0 <= j && j < x.length;
8         @           x[j] == \result); */
9       static int max(int[] x) {
10          int i = 1;
11          int max = x[0];
12          while (i < x.length) {
13              if (max < x[i]) {
14                  max = x[i];
15              }
16              ++i;
17          }
18          return max;
19      }
20  }
```

Though the precondition conjunct `(x != null)` is the default in JML, it is useful to include it explicitly to further the discussion of `null` replacement. Moreover, approaches to conform to the Rapid's limited grammar (see listing 3.1) are illustrated in the following subsections such as handling method parameters and an `else` clause.

4.1.1 Translating from Java into Rapid

In this section, an approach to the translation from Java to Rapid is addressed.

4.1.1.1 Translating Method Parameters from Java to Rapid

The grammar of Rapid's programs is simpler than that of Java programs as described in listing 3.1. For example, a Rapid program only contains one function, `main`, and it does not take any function parameters. The problem is translating a Java program so that a translated program does not have any function parameters. To solve this problem, I remove all the method parameters in Java programs and locally declare them inside the function's block.

Moreover, since a length of an array is often used in specifications and a program, the translation must also declare a constant of the array's length if a declared variable has an array type.

For example, the `max` method in program 4.1 contains one integer array parameter: `int[] x`. This integer array will be a local variable array in the translated program and would look like below:

```
1    func main() {
2        const Int[] x;
3        const Int xlength;
4        // rest of the translated code
5    }
```

Before translating a program, my tool preprocesses the program decides if variables are constants or not. if they are updated in the program, they are classified as constants and declared with `const` modifier in a translated program. Otherwise, they are declared without

any modifiers. As the array`x` and its elements are not updated in the program, the array `x` is declared as a constant in the translated program.

4.1.1.2 Translating Other Statements

Another problem of the Rapid grammar is that `if` statements must have an `else` clause though they need not in Java. To solve this issue, I use the `skip` statement in Rapid and add an `else` clause that only contains the `skip` statement if an original `if` statement in the Java program do not contains an `else` clause.

Consequently, the if statement in line 13 in listing 4.1 will be followed by an `else` clause in the translated Rapid program:

```
1       if (max < x[i]) {
2           max = x[i];
3       } else { skip; }
```

In Rapid, unary expressions such as decrement or increment of integers are not supported. However, it is simply replaceable with an equivalent assignment statements. For example, an increment of variable `i`, *i.e.* `i++` is equivalent as the following expression.

```
i = i + 1;
```

4.1.1.3 The Limitation of Translation

Because of the limited and simple grammar of Rapid, programs which are not expressible in Rapid cannot be directly translated. For example, exception handlings using `try-catch` expressions are not in the Rapid's syntax. In this case, a user should convert an original program into one that has the similar semantics by using `if-else` statements to conform to Rapid's grammar. This approach will be left for future work.

4.1.2 Translating JML into the Supported SMT-LIB Languages

To use Rapid, a user has to provide it with a program and its safety property. The safety property of the program is written in the SMT-LIB language. It is constructed by an implication from the precondition to the postcondition of the Java program. In this section, I consider translating preconditions and postconditions written JML into the supported SMT-LIB language.

4.1.2.1 Null Deletion for JML to SMT-LIB Translation

The SMT-LIB language does not support all Java programming concepts. An important example of this problem is that the literal `null` has no counterpart in SMT-LIB, and thus the translation of JML to SMT-LIB must eliminate uses of `null`. Our approach to this

problem is deleting all the expressions containing `null` appearing in JML preconditions and postconditions before translating them. This means that the semantics of an expression containing `null` will be disregarded completely. If a JML method clause expression is entirely an expression containing `null`, such clause expression will be completely disregarded and deleted from the list of method specifications. For instance, suppose that a JML precondition only specifies that an array should not be null before method execution; `(//@ requires x != null;)`. This precondition will be deleted entirely and will not appear in the translation.

4.1.2.2 Translation of the Keyword result in JML

The keyword `\result` in JML means a method's return value. The grammar in Rapid does not have a return statement, so the function does not return anything in Rapid. However, Rapid has a concept of timestamp [10] and a user can specify a value of a variable at a certain point. The general rule for translating `\result` is that every variable used in a return expression is considered as a value at the end of the program.

For example, `(max main_end)` means the value of the variable `max` at the end of the function `main`. Therefore, in the program 4.1, I can replace `\result` with `(max main_end)`.

4.1.2.3 Variable Name Conflict in Quantifier Expressions

In Java/JML programs, a user can shadow variables that appear in the program in program specifications written in JML. In the safety property in Rapid, a user can refer to immutable variables *i.e.,* constants in a program. However, when referring to mutable variables, a user must write a timestamp to specify which value of the variable in the program such as `(max main_end)`. Therefore, mutable variables such as loop variants cannot be referred to in the safety property in Rapid without timestamps. When translating a JML specification that contains mutable variables in the program, I change their names by adding a number at the end of the variable names and use them in the safety property to prevent name conflicts.

For example, the postcondition of program 4.1 contains a bound variable `i` in its universal quantifier expression. However, when a user uses this variable `i` in a safety property, Rapid thinks of it as a loop variable `i` in the program. To prevent such conflict, I keep adding an integer number starting from 0 at the end of `i` until the new variable name does not conflict with any other variable names. Therefore, first, I add 0 at the end of `i`. Since the variable name `i0` is not used anywhere in the program, I use `i0` in the translated safety property. The resulting safety property will look like this:

```
(forall ((i0 int))
    (=>
        (and (>= i0 0 ) (< i0 xlength))
        (<= (x i0) (max main_end))
    )
```

)

4.2 Translating from SMT-LIB into JML

In this section, I describe approach to the translation problem from SMT-LIB into JML.

4.2.1 Translation of Operations

In SMT-LIB, a user can write simple arithmetic operations and logical operations. The major difference between JML and them is that an operator comes first in an expression written in SMT-LIB. For example, a simple preposition `(5 == (2 + 3))` in JML is written as `(= 5 (+ (2 3)))`. Therefore, when translating an arithmetic/logical expression, the left hand side of the expression must be translated prior to an operator.

4.2.2 Translation of Special Constants

When a variable is not constant, Rapid/Vampire defines two new constants with `_init` and `_final` suffixes with respect to the variable. A constant with `_init` suffix refers to the variable's initial value. A constant with `_final` suffix refers to the value of the variable at the end of the program. For instance, Rapid/Vampire declares a constant `max_init` as the initial

value of the variable `max` in program. To enable using this variable in JML specifications, I define a same constant initialized with `max`:

```
//@ ghost final int max_init = max;
```

By injecting this specification right before `max`'s initialization, `max_init` can refer to its initial value. The `ghost` modifier in JML allows a user to define properties only used in specifications. Now that I have `max_init` defined, `max_init` can freely be used in specifications without being translated into something else.

As a variable can be updated after a loop and by the end of the program, using `_final` constants in loop invariants is not useful. Therefore, all the propositions that contain a constant with `_final` suffix will not be included as candidate loop invariants and will not be translated.

Another special variable to consider is a length constant of an array. As mentioned in 4.1.1.1, every time the translator encounters an array, its length is also declared as constant in a manner of concatenating 1) the name of an array and 2) the keyword `length`. As a field access is allowed in JML, it is translated back into normal Java syntax. Therefore, `xlength` is translated into `x.length` in JML specifications.

4.2.3 Translation of `declare-fun` and `declare-const`

Loop invariants generated by Rapid/Vampire contain two commands to declare functions and constants used in specifications.

In SMT-LIB, a user can use `declare-fun` to declare a new function, and `declare-const` to declare a new constant used in specifications. In JML, a user can use the `model` modifier to declare new methods or fields that are only available in specification purposes. Note that `declare-fun` declares a new function but does not define its implementation. Therefore, a translated model method does not contain its implementation, either. For example, the following two commands are an array `x` written in SMT-LIB.

```
(declare-fun x (Time Int) Int)
```

they are translated into

```
//@ public pure model int x(Time a, int b);
```

where `a` and `b` are alphabetically generated method parameters.

Thanks to preprocessing of generated loop invariants, all the functions or constants that are not used in specifications will not be declared in translated specifications.

CHAPTER 5
RELATED WORK

In this chapter, I discuss some work related to automatically generating a loop invariant.

5.1 LoopGen

Saswat Padhi at el. [11] developed a tool to infer a loop invariant as an extension of their previous work [12]. Their approach reduces the loop invariants inference problems to a series of precondition inference problems. It is able to infer a loop invariant of a program that contains while loops. However, the programs that this paper generates loop invariants for are written in the syntax-guided synthesis (SyGuS) language. Therefore, it does not generate loop invariants for Java programs. To use this tool as a loop invariant generator for OpenJML, we would need to translate Java programs to the supported SyGus language.

5.2 C2I

Rahul Sharma and Alex Aiken proposed a framework called C2I which generates a loop invariant inference procedure from an invariant checking procedure [13]. The generated inference procedure invokes two phases, the search phase and the validate phase. The search phase uses randomized search to find candidate loop invariants. Then, the second phase

verifies the candidate loop invariants found in the search phase. However, the goal of that paper is to provide a framework to generate a invariant inference engine, not to infer a loop invariant of a program that contains loops.

5.3 Houdini

Cormac Flanagan and K. Rustan M. Leino created a tool called Houdini to perform modular checking which verifies a part of a code written in Java [14]. Houdini generates a large number of candidate annotations for a module and use ESC/Java [15] checker to accept or refute them. However, annotations that it generates are dependent on an initial candidate annotation set, and those annotations are invariants for any field, or preconditions and postconditions for a method in a program. Hence, though that paper's target programs are written in Java, the objective is not to find a loop invariant. It might be useful to use this tool for the purpose of assisting to generate a safety property of a program before translating Java/JML programs into Rapid.

5.4 HOLA

Isil Dillig at el. proposed a new method to generate inductive loop invariants [16]. Their approach is that they strengthen loop invariants starting from `true` until they are strong enough to prove the correctness of a program. They provided their own imperative language

Table 5.1: List of BasicBlock nodes

JML assume statements	JML assert statements	JML comment statements
JCLiteral	JCIdent	JCTypeCast
JCParens	JCUnary	JCBinary
JCConditional	JmlBBFieldAccess	JmlBBArrayAccess
JmlBBFieldAssign	JmlBBArrayAssign	JCMethodInvocation
JmlMethodInvocation	JmlQuantifiedExpr	

to formalize their technique. Therefore, if we integrate this tool to OpenJML, we would need to translate Java/JML programs into the supported language. Unfortunately, this tool is not available as of now, so I could not try it as an alternative.

5.5 SMTTranslator

A class `SMTTranslator` in `org.jmlspecs.openjml.esc` package translates BasicBlock program into SMT-LIB. This BasicBlock program consists of the limited subset of Java AST nodes. The list of the supported BasicBlock nodes is in table 5.1. `SMTTranslator` does not translate many of the Java AST nodes, especially an array access and the keyword `\result`. However, it already provides a core platform of translation from JML into SMT-LIB. In current implementation, `SMTTranslator` prints out an error message if it visits an unsup-

ported node. It would be promising if one can extend `SMTTranslator` and provide correct translations for unsupported nodes instead of printing an error message.

CHAPTER 6
DISCUSSION

In this chapter, I describe the evaluation of loop invariants generated by Rapid/Vampire. I used the latest Rapid of a main branch as of May 18th 2022, which is available on Github[1]. For Vampire, I used a custom version of Vampire[2]. The loop invariant generation is performed on an Intel i5 2 GHz CPU with 8 GB of memory.

6.1 Loop Invariants Generated by Rapid/Vampire

The translated program from program 4.1 for Rapid is the following:

```
1  func main()
2  {
3    const Int[] x;
4    const Int xlength;
5    Int i = 1;
6    Int max = x[0];
7    while (i < xlength) {
8      if (max < x[i]) {
9        max = x[i];
10     } else { skip; }
11     i = i + 1;
```

[1]https://github.com/vprover/rapid
[2]https://github.com/vprover/vampire/tree/michael-rapid-symel-chimera

```
12    }
13  }
14  (axiom
15    (> alength 0)
16  )
17  (conjecture
18    (forall ((i0 Int))
19      (=>
20        (and (>= i0 0) (< i0 xlength))
21        (<= (x i0) (max main_end))
22      )
23    )
24  )
25  (conjecture
26    (exists ((j Int))
27      (and
28        (and (<= 0 j) (< j xlength))
29        (= (x j) (max main_end))
30      )
31    )
32  )
```

According to a discussion with developers of Rapid/Vampire, I used configurations that they recommend. For running Rapid, I used the following option:

```
-invariantGeneration on
```

Rapid generates multiple outputs written in SMT-LIB which describe program semantics and axioms. Among these files, `user-conjecture-*.smt2` files are what I am interested in.

For running Vampire, I used the following options:

```
-t 1
--input_syntax smtlib2
--output_mode vampire
--statistics none
--symbol_elimination on
-sa otter
-av off
-fde none
-updr off
-erd off
-p off
```

The proposed approach could successfully integrate Rapid/Vampire into OpenJML and automatically generate candidate loop invariants for a while loop in a program written in JML.

Though Rapid/Vampire automatically generates loop invariants, there are many candidates properties that it produces so we need an extra step to pick up useful ones out of them. For example, Rapid/Vampire generated, in total, 315 candidates for loop invariants of program 4.1. Some of them are very trivial and not useful to prove the loop's correctness. The following code is an example of the trivial propositions generated by Rapid/Vampire.

```
-6 < x.length;
```

This means that the length of the array `x` should be greater than -6, which does not help to prove anything because the length of `x` is already asserted to be greater than 0.

Also, some properties are not also useful because they are tautologies in the relevant theory such as the following preposition which describes that an integer i minus 1 is always smaller than i.

```
\forall int i; i + 1 > i;
```

The desired loop invariant for the first postcondition of program 4.1 should be the following proposition:

```
\forall int k; 0 <= k && k < i; x[k] <= max;
```

However, after manually inspecting each candidate loop invariants generated by Rapid/Vampire, none of them was the desired loop invariant or infers the desired loop invariant.

To automatically check which candidate loop invariants are used to prove postconditions, I ran Vampire with `casc` mode on a file such that I concatenated the generated loop invariants to one of the Rapid's output file that describes semantics of postcondition, namely `user-conjecture*_postcondition.smt2`. Specifically, I used the following option:

```
--mode casc
```

The result shows that Vampire could not find any proof to prove the postcondition from the generated loop invariants.

Therefore, one needs to use a different loop invariant generator to integrate it with OpenJML. Possibly useful tools would be the ones discussed in chapter 5. Vampire provided a feature of program analysis [17] which runs on a program written in a subset of C, it is not implemented anymore in the available version of Vampire unfortunately.

CHAPTER 7
CONCLUSIONS

In this thesis, to address the challenge that JML does not support generating loop invariants for Java code, I proposed two translation packages to integrate a loop invariant generator tool. A translation package from Java/JML programs into programs written in Rapid's syntax and SMT-LIB has been successfully integrated with OpenJML. A translation package from SMT-LIB into JML has been integrated with jSMTLIB. By using these two translation programs, I could automatically generate loop invariants of a loop program written in Java. However, generated loop invariants are not useful to prove the correctness of the program. Therefore, a further support of an external tool will be necessary to successfully generate loop invariants from Java programs. As my translation packages are designed as extensible, a user can utilize my translations to support other potential loop invariant generation tools with little implementation effort.

LIST OF REFERENCES

[1] P. Georgiou, B. Gleiss, and L. Kovács, "Trace logic for inductive loop reasoning," in *2020 Formal Methods in Computer Aided Design (FMCAD)*, pp. 255–263, IEEE, 2020.

[2] K. Hoder, L. Kovács, and A. Voronkov, "Invariant generation in vampire," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 60–64, Springer, 2011.

[3] D. Cok, "jSMTLIB." `https://github.com/smtlib/jSMTLIB`. [Online; accessed 04/05/2022].

[4] E. W. Dijkstra, "Guarded commands, nondeterminancy and formal derivation of programs," *Communications of the ACM*, vol. 18, pp. 453–457, Aug. 1975.

[5] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, pp. 576–580,583, Oct. 1969.

[6] A. M. Turing *et al.*, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[7] C. Bruni, "Program Verification While Loops." `https://cs.uwaterloo.ca/~cbruni/CS245Resources/lectures/2018_Fall/19_Program_Verification_While_Loops_post.pdf`. [Online; accessed 05/04/2022].

[8] R. C. Jeffrey, *Formal Logic: Its Scope and Limits*. New York, NY: McGraw-Hill Book Co., 1967.

[9] "Rapid Github page." `https://github.com/vprover/rapid`. [Online; accessed 05/09/2022].

[10] B. G. Pamina Georgiou, "Rapid Semantics." `https://github.com/vprover/rapid/blob/main/doc/semantics/main.pdf`. [Online; accessed 05/08/2022].

[11] S. Padhi, R. Sharma, and T. Millstein, "Loopinvgen: A loop invariant generator based on precondition inference," *arXiv preprint arXiv:1707.02029*, 2017.

[12] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," PLDI '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[13] R. Sharma and A. Aiken, "From invariant checking to invariant inference using randomized search," *Formal Methods in System Design*, vol. 48, no. 3, pp. 235–256, 2016.

[14] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for ESC/Java," in *FME 2001: Formal Methods for Increasing Software Productivity* (J. N. Oliveira and P. Zave, eds.), vol. 2021 of *Lecture Notes in Computer Science*, pp. 500–517, Springer-Verlag, Mar. 2001.

[15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for Java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, vol. 37(5) of *SIGPLAN*, (New York, NY), pp. 234–245, ACM, June 2002.

[16] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," *Acm Sigplan Notices*, vol. 48, no. 10, pp. 443–456, 2013.

[17] K. Hoder, L. Kovács, and A. Voronkov, "Case studies on invariant generation using a saturation theorem prover," in *Mexican International Conference on Artificial Intelligence*, pp. 1–15, Springer, 2011.