Formal Program Verification in Java

Marc Etter
OST Eastern Switzerland University of Applied Sciences
MSE Seminar “Programming Languages”
Supervisor: Farhad Mehta
Semester: Autumn 2021

Abstract

Formal verification is usually taught with a focus on functional languages, such as Agda, or specialized languages, such as Dafny. However, it remains a fact that these languages are not very commonly used in mainstream software development. Nevertheless, strong guarantees about correctness of software remain highly desirable in nearly all applications. This paper attempts to bridge the gap between formal verification methods and mainstream software development. To achieve this goal, the paper presents state-of-the-art tools and technologies that can be used to formally verify Java programs. Futhermore, a focus is set on ease-of-use and learning simplicity, as a measure for how viable a tool is for use during teaching at the bachelor's level.

Keywords: Formal Program Verification, Java, JML, OpenJML, KeY

1 Introduction

Program verification in software engineering is the practice of mathematically proving that a program conforms to a formal specification. Optimally, this is done statically, i.e. without the need to execute the program to be verified. As program verification aims to prove the correctness of a program, it is similar to manual or automated testing in that regard. However, the big difference is that the effort in testing grows exponentially, as the confidence in the test results usually only asymptotically approaches 100%. On the other hand, program verification often has a higher base effort, but the effort for very high confidence-levels is usually significantly lower, compared to testing.

For this reason, formal program verification is a desirable method for use in critical applications, where a high confidence in the correctness is required. Currently, many developers prefer to use specialized programming languages which lend themselves better to formal verification. However, these programming languages are generally less well-known and represented in the field. Therefore, the risk to a company grows, as they are reliant on developers of niche skills.

Java is one of the most popular programming languages in mainstream software development. Many companies and developers choose to work with or learn Java, simply for the reason that it is easy to find developers and jobs in Java. Over time, Java has evolved into a complex language with features that make it hard to verify. Thus, Java is not a language known for its formal verification practices.

Nevertheless, there are multiple projects attempting to bring the world of formal program verification into the Java ecosystem, and closer to the Java developer. This paper presents an introduction to some of the more popular options in this field.

1.1 Assumed Knowledge

This paper assumes the reader has a good understanding of the Java programming language. Basic principles of formal verification, such as Hoare-logic, weakest preconditions, and loop invariants are helpful, but not strictly required [Hoa69].

2 Formal Verification Tools for Java

When attempting to verify a program, there are multiple components involved in that process, as illustrated by Figure 1.

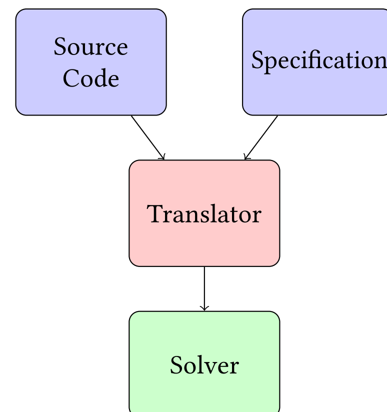

Figure 1. Toolchain for Formal Verification of a Program

This paper focuses on Java, therefore the source code will be Java code. The specification is usually a custom language, suited to write formal, machine- and human-readable specifications close to the code. Translators take the formal specification and transform them to logical and mathematical verification conditions that need to be solved. Finally, those proof obligations are passed to a solver software that attempts to solve them, which is equivalent to proving the correctness of the code under its specification. Analyzing those theorem solvers is out-of-scope for this introductory paper; the default for each tool will be used. Therefore, this paper will focus on the specification language and translators.

2.1 Java Modelling Language (JML)

The Java Modelling Language is a specification language used to specify the behavior of Java code [LBR99]. It is written inside Java comments, prefixed with an `@` symbol. JML itself only provides the syntax and semantics of how to write formal specifications of Java code, but it provides no tool for verification itself. Such tools are presented in the later sections of this paper.

A simple example of a JML specification might look as shown in listing 1.

```
1 public class Math {
2     /*@ requires x >= 0 && x <= 10;
3       @ ensures \result * \result <= x;
4       @ ensures (\result + 1) * (\result + 1) > x;
5      */
6     public static int intSqrt(int x) { /* ... */ }
7 }
```

Listing 1. Simple JML specification for a single method

The `requires` clause declares a precondition that must hold for the caller to be allowed to call this method. Similarly, `ensures` specifies a postcondition that is guaranteed to be true after the method exits. In this example, the valid argument range is limited to the interval $[0, 10]$. This is done for demonstration purposes, but might be relevant, for example, if the implementation relies on a pre-computed table of results.

Even when specifying JML for seemingly trivial methods, faults that can easily get overlooked are immediately found by a verifier.

```
1 public class Math {
2     //@ ensures \result == x * x;
3     public static int square(int x) {
4         return x * x;
5     }
6 }
```

Listing 2. Seemingly trivial but incorrect implementation

Attempting to verify the example in Listing 2 with OpenJML (see section 2.2) results in an `ArithmeticOperation\-Range` warning "int multiply overflow". We have naively overlooked the scenario that our input argument could be large enough for the multiplication to result in an integer overflow. This can be fixed by either adding a precondition with an upper limit on the input, or by casting `x` to `long` before the multiplication and returning the `long`.

```
1 public class CopyArray {
2     /*@ requires a.length == b.length;
3       @ requires begin >= 0 && begin < a.length &&
            end >= 0 && end < a.length && begin <=
            end;
4       @ ensures (\forall int i; begin <= i && i <
            end; a[i] == b[i]);
5      */
6     public static void copyArray(int[] b, int begin,
          int end, int[] a) {
7         /*@ maintaining begin <= k && k <= end;
8           @ loop_invariant (\forall int i; begin <=
              i && i < k; a[i] == b[i]);
9           @ decreases end - k;
10          */
11        for (int k = begin; k < end; k++) {
12            b[k] = a[k];
13        }
14    }
15 }
```

Listing 3. JML specification for a copyArray method

A slightly more complex example is shown in Listing 3. There, a method for copying the contents of one array into another array is presented, together with the corresponding JML specification. The `CopyArray` example shows that JML is not only used on methods, but can also be used inside methods, for example on loops. JML makes use of the familiar concept of loop invariants together with a decreasing condition to prove loop termination and correct loop results.

One alternative to JML is *Contracts for Java* (Cofoja) [Lê11]. Although it uses similar syntax, Cofoja is intended as a specification language from which runtime checks are generated into the application. This paper focuses on static program verification. Therefore, Cofoja will not be covered hereafter.

2.2 OpenJML

OpenJML is a suite of tools developed mainly by David Cok [Cok14]. OpenJML's website[1] lists three main capabilities:

- parsing and type-checking
- static checking
- runtime-assertion checking

Within the scope of this paper, the runtime-assertion checking of OpenJML will be disregarded. Parsing and type-checking are usually already covered by the IDE - at least for the actual Java code.

Installation and usage of OpenJML is very straight-forward. A jar-file can be downloaded from the website and immediately run on the command-line. With the upcoming version built with JDK 16, an executable file will be released with the required JRE pre-packaged [Cok21]. This obsoletes the need to locally install the correct JRE or JDK for both OpenJML and the application being developed. Therefore, potential complications if different versions are used can be avoided.

The `-esc` command-line switch enables "extended static checking", which includes verification of JML annotations. For the purpose of this paper, this is the only relevant command-line option, other than `-sourcepath` to let OpenJML know where the local source files are located. To make the usage of OpenJML even easier, most IDEs support run configurations or tasks. With clever setup, verifying a file can be as simple as pressing a button or hotkey in the IDE. An example setup of a run configuration for IntelliJ IDEA is shown in figure 2, which will verify the currently open file when run. For concrete examples on how to use OpenJML, see section 3. Furthermore, appendix B lists a dockerfile for running the

[1] https://www.openjml.org/ (2021)

newest version of OpenJML based on JDK 16, for which there is no Windows build available yet.

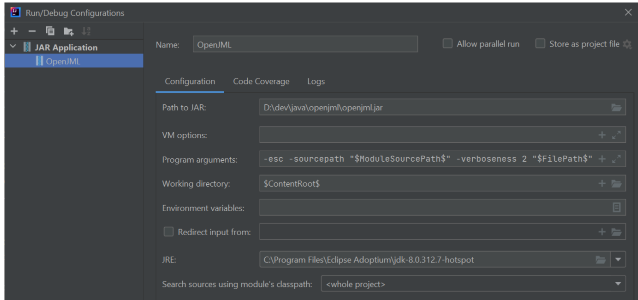

Figure 2. IntelliJ run configuration for OpenJML

However, OpenJML is in principle not interactive. This means if a solver is struggling to verify some condition, it is not possible for the developer to intervene and help the solver out. Instead, all necessary hints must be written in JML, or the solver will fail. Furthermore, some experience is required to understand the root cause of why a proof-attempt failed. For example, a warning issued by OpenJML might reference a post-condition that could not be verified, when really the pre-condition might not have been accurate. Lastly, OpenJML has limited support for reasoning about floating-point logic. However, most tools share this limitation.

The developers of OpenJML provide an extensive suite of JML specifications for JDK classes on their GitHub page[2]. This allows code to be verified that makes use of JDK functionality, without the user having to provide those specifications. Furthermore, development on OpenJML is very active. Raised issues, such as bug reports, are answered and fixed within a couple of days. Additionally, an IntelliJ plugin for OpenJML has been published[3], although it does not yet support the full feature-set of JML [MSW+21].

2.3 KeY

The KeY project was started in 1998 at the *Karlsruhe Institute of Technology* by Hähnle et al. [HMS98]. According to *The KeY Book*, "[..] the aim of the KeY project was to integrate formal software analysis methods [..] into the realm of mainstream software development." [ABB+16].

Similar to OpenJML, KeY is easy to get started with. The KeY website[4] provides a download to an executable JAR-file. That JAR-file then launches a graphical user interface, into which the user can load their source files. Furthermore, the application comes packaged with examples, ready to be loaded and tested.

By providing a graphical user interface, the application might be more approachable for new students. However, the application is definitely tailored to expert usage. Therefore, a new user should be guided closely, otherwise they might be overwhelmed by the amount of buttons, windows, and general information provided. That being said, KeY visualizes every step the solver takes when attempting to verify some program. Although not trivial to understand, this can be an important tool in teaching, where a teacher can show students exactly what happens under the hood: all the theory about formula simplifications and transformations can be seen in action. Furthermore, KeY supports interactive proofs; a procedure where the user can intervene when a solver gets stuck, and try to apply manual transformations to guide the solver. Again, this can be both challenging but also enlightening. KeY also seems to be in relatively active development, with the latest stable release published in late 2020.

While KeY seems to be promising in most aspects, it adds more complexity for the user than for example OpenJML. With KeY, the user not only needs to learn how to write correct JML, they must now also learn how to operate the KeY tool. Furthermore, at the time of writing, there do not seem to be any efforts on updating KeY to Java versions newer than 8. However, in regards to teaching, the KeY project does provide a complete book covering JML, formal methods in general, and the KeY application [ABB+16].

2.4 Krakatoa

Krakatoa is a tool developed by the *Laboratoire de Récherche en Informatique* in Paris as a front-end to the Why platform [DR19]. Krakatoa takes a KML-annotated Java program (a variant of JML) and transforms it into Jessie; an intermediate language common to Java and C. This intermediate Jessie language can then be passed to Why's verification condition generator, and ultimately forwarded to the provers. This architecture is visualized in figure 4.

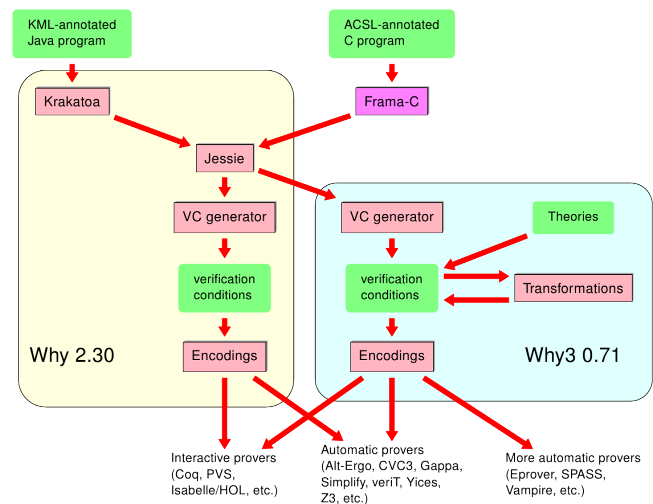

Figure 4. Why Architecture [Lab15]

[2] https://github.com/OpenJML/Specs (2021)
[3] https://gitlab.utwente.nl/fmt/intellijml (2021)
[4] https://www.key-project.org

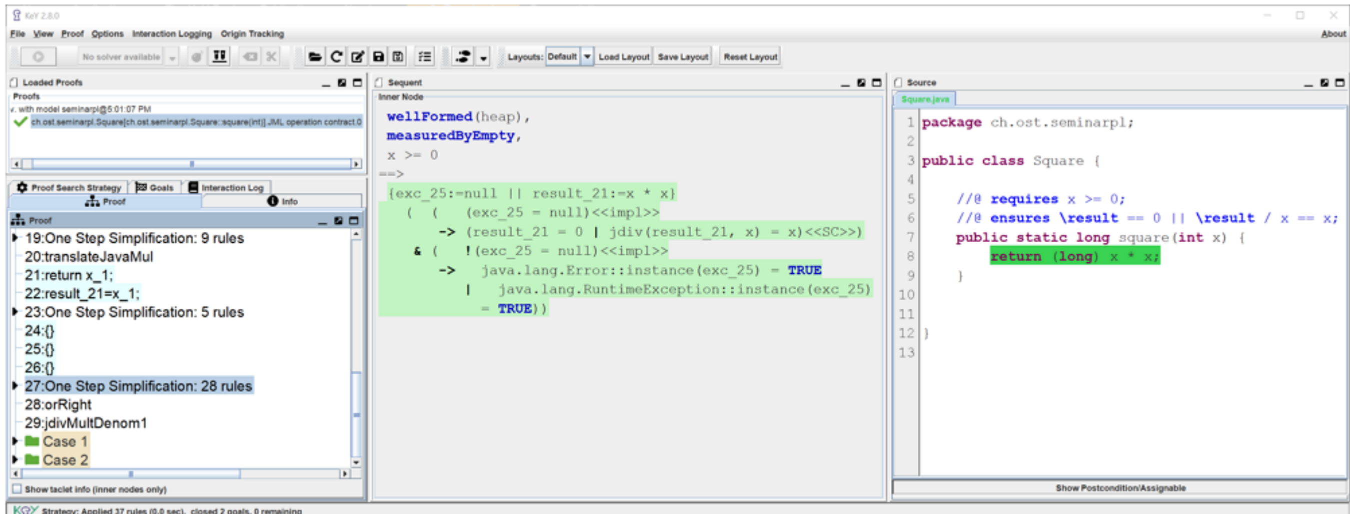

Figure 3. An automated proof being visualized in KeY

At the time of writing, Why3 was still in its early development stages. According to the Krakatoa website, it is recommended to install Why3 conjointly with Why version 2.41 [Lab15]. Unfortunately, the author was not able to get the setup to work after about a day of effort. Seeing as further development on Krakatoa and Why are currently frozen, this paper will not delve deeper into Krakatoa. A new evaluation at a later date, when development of Why3 has progressed far enough for Krakatoa to be on-boarded, would be necessary. According to a case study by Dmitry Brizhinev, Krakatoa seems to be very promising, if one can get it to run [BG18]. In particular, it seems to be one of the few tools supporting accurate floating point verification. Similarly, Divasón and Romero concluded that Krakatoa was the best tool for bachelor's level teaching [DR19], albeit hard to set up.

3 Example: IntStack with OpenJML

To get a better understanding of JML and formal verification in general, this section presents an exercise from Prof. Farhad Mehta's bachelor course "SE Practices 1", which uses Cofoja. The provided source code will be extended step-by-step with JML and verified using OpenJML.

In this example, we try to formally specify (and subsequently verify) a simple stack of integers. The interface for such an `IntStack` is presented in listing 4.

```
1  interface IntStack {
2      void push(int value);
3      int pop();
4      int top();
5      boolean isEmpty();
6      boolean isFull();
7  }
```

Listing 4. Bare interface of the `IntStack`

Without worrying about the implementation for now, we can already enhance this interface with JML specifications. These specifications cannot be verified yet, until we provide an implementation of `IntStack`. Listing 5 shows the `IntStack` interface extended with simple `requires` and `ensures` directives. This should already be sufficient to ensure the basic integrity of the stack: no elements may be added when full, and no elements may be removed when empty.

```
1  public interface IntStack {
2
3      /*@ requires !isFull();
4        @ ensures !isEmpty();
5        @ ensures top() == value;
6       */
7      void push(int value);
8
9      /*@ requires !isEmpty();
10       @ ensures !isFull();
11       @ ensures \old(top()) == \result;
12      */
13     int pop();
14
15     /*@ requires !isEmpty();
16       @ pure
17      */
18     int top();
19
20     //@ pure
21     boolean isEmpty();
22
23     //@ pure
24     boolean isFull();
25 }
```

Listing 5. `IntStack` with basic JML

Before moving on to the implementation, we can use OpenJML to verify that our JML is correct in regards to syntax and basic semantics (e.g. there are no contradictions). Listing 6 shows the command to achieve this.

```
1  $> java -jar openjml.jar \
2      -sourcepath src/main/java \
3      -esc src/main/java/intstack/IntStack.java
```

Listing 6. Verify `IntStack` using OpenJML

Although adding JML to an interface is a nice exercise to think about the behavior of something, this does not provide us with anything concrete to verify yet. Listing 7 shows an example of a simple array-based implementation of `IntStack`.

```
class IntStackArrayImpl implements IntStack {
    private final int[] values;
    private int topIndex = -1;

    public IntStackArrayImpl(int size) {
        values = new int[size];
    }

    @Override
    public void push(int value) {
        values[++topIndex] = value;
    }

    @Override
    public int pop() {
        return values[topIndex--];
    }

    @Override
    public int top() {
        return values[topIndex];
    }

    @Override
    public boolean isEmpty() {
        return topIndex == -1;
    }

    @Override
    public boolean isFull() {
        return topIndex == values.length - 1;
    }
}
```

Listing 7. Array-based implementation of `IntStack`

Without any JML on the implementation itself, we can already check the specification of the interface against our implementation. The command in Listing 8 will run a static verification using OpenJML.

```
$> java -jar openjml.jar \
    -sourcepath src/main/java \
    -esc src/main/java/intstack/IntStackArrayImpl.
        java
```

Listing 8. Verify `IntStackArrayImpl` using OpenJML

Running OpenJML on the bare `IntStackArrayImpl` should yield a couple of warnings: possibly negative array indices, possibly too large array indices, and integer overflow/underflow. Furthermore, OpenJML will complain about some postconditions that cannot be verified - these are the `ensures` directives of the interface.

To resolve these warnings, we must either change the implementation, or add additional JML declarations to it. For the purpose of this example, we will not modify the Java code.

3.1 Constructor

The constructor only has one warning: `PossiblyNegativeSize`. This is easily fixed as shown in listing 9 by using a precondition.

```
//@ requires size >= 0;
public IntStackArrayImpl(int size) {
    values = new int[size];
}
```

Listing 9. Fix `PossiblyNegativeSize` with requires

3.2 Invariants

To get rid of all `PossiblyTooLargeIndex` warnings (except for the one in `push()`), we can introduce an invariant that ensures that `topIndex` never exceeds the array's size, as shown in listing 10. This invariant must be prefixed with the visibility qualifier `private`, as the invariant references fields that are private. In other words, this invariant is not observable from outside the class itself. While adding an invariant for the upper bound of the `topIndex`, we might as well also add the invariant for the lower bound.

```
class IntStackArrayImpl implements IntStack {
    //@ private invariant topIndex < values.length;
    //@ private invariant topIndex >= -1;

    // ... rest omitted
}
```

Listing 10. Class invariants for `IntStackArrayImpl`

3.3 Private Behavior

After adding the class invariants, all remaining OpenJML warnings should relate to `push`, `pop`, and `top`. All three of these methods are specified in terms of `isEmpty` and `isFull` on the interface. Unfortunately, OpenJML does not sufficiently inspect the implementation of the boolean methods, therefore we need to provide some hints. Listing 11 shows the necessary JML. The JML for `isFull` is very similar.

```
/*@ also
  @   private normal_behavior
  @     ensures \result == (topIndex == -1);
  */
@Override
public boolean isEmpty() { /* ... */}
```

Listing 11. Private behavior for `isEmpty`

By adding these directives, we let JML know that the `isEmpty` method has a clearly defined "normal behavior" (i.e. no exceptions), but this behavior is not observable from the outside (private).

Finally, the last two warnings relate to what the `top` of the stack is before and after the `push` and `pop` operations. Similar to the boolean methods, a private normal behavior resolves this issue, as shown in listing 12

```
/*@ also
  @   private normal_behavior
  @     requires !isEmpty();
  @     ensures \result == values[topIndex];
```

```
5    */
6   @Override
7   public int top() { /* ... */ }
```

Listing 12. Private behavior for `top`

3.4 Usage

Although we resolved all OpenJML warnings on our implementation, our work is not quite complete yet: When trying to use the `IntStack` as shown in listing 13, OpenJML tells us that the `push(2)` call (amongst others) cannot be verified. The reason is that the precondition of the stack not being full cannot be verified. In other words, OpenJML currently cannot know that after one push-call, the stack is not full yet, even though we created a stack with maximum capacity of 2.

```
1   IntStack stack = new IntStackArrayImpl(2);
2   stack.push(1); // {1}
3   stack.push(2); // {1, 2}
4   stack.pop();   // {1}
5   stack.push(3); // {1, 3}
6   stack.pop();   // {1}
7   stack.pop();   // {}
```

Listing 13. Usage example of `IntStack`

3.5 Model variables

To fix this, we need to add the concept of stack size and capacity into JML. This could relatively easily be solved by adding corresponding new methods to the interface. However, this section will detail an approach solely using JML.

We start by declaring model variables on the interface to track size and capacity. Model variables are abstractions of real Java variables. However, model variables are only available in the JML context for use during verification. In the case of the `IntStack`, we need the maximum capacity and the current size. Furthermore, we declare invariants for those model variables. Listing 14 shows the complete declaration of the model variables.

```
1   interface IntStack {
2     /*@ public model instance int stackSize;
3       @ public model instance int capacity;
4      */
5
6     /*@ public instance invariant stackSize >= 0;
7       @ public instance invariant stackSize <=
            capacity;
8      */
9
10     // ... rest omitted
11  }
```

Listing 14. Model variables declaration

The `instance` keyword is required when declaring model variables on an interface. It tells JML that this model variable exists on every instance of the interface, as opposed to on the `class` object, if it were declared as `static`.

Now that we have more concrete variables to work with, we can rewrite the JML annotations for most methods. The extended JML for the `push` method is shown in Listing 15.

```
1   /*@ requires stackSize < capacity;
2     @ assignable \everything;
3     @ ensures stackSize == \old(stackSize) + 1;
4     @ ensures capacity == \old(capacity);
5     @ ensures top() == value;
6    */
7   void push(int value);
```

Listing 15. `push` specification with model variables

The precondition for the methods are still the same, but now expressed in terms of the exact `stackSize` and `capacity`. The `assignable \everything` directive is a new addition, telling JML that all variables may change as a side-effect of this method. Normally, one would specify exactly which variables may be changed as a side-effect. However, OpenJML does not yet support the `maps` directive, which would allow the implementation to specify that the array contents can be modified[5]. As a workaround, we can use `\everything` together with an `ensures` to tell JML that the `capacity` may not change when calling this method. The extended JML annotations for the other interface methods can be found in appendix A.

At this point, we get many warnings from OpenJML about the model variables. This is because we have not told JML yet what those model variables are an abstraction of. To do this, we need to add `represents` directives to the implementation, as shown in Listing 16. The same listing also shows small extensions to the JML annotations on the constructor and instance fields.

```
1   private final int[] values; //@ in capacity;
2   private int topIndex = -1; //@ in stackSize;
3
4   /*@ private represents stackSize = topIndex + 1;
5     @ private represents capacity = values.length;
6    */
7
8   /*@ requires size >= 0;
9     @ ensures stackSize == 0;
10    @ ensures capacity == size;
11   */
12  public IntStackArrayImpl(int size) { /* ... */ }
```

Listing 16. `represents` declarations in `IntStackArrayImpl`

As an added bonus, the private behaviors added previously for `isEmpty` and `isFull` are no longer necessary. With these model variable changes done, the interface, the implementation, as well as the usage all verify successfully with OpenJML[6].

By declaring model variables on an interface with the `represents` clauses in the implementation, we basically used subtype polymorphism within JML. Every implementation of the `IntStack` interface must now declare itself how the concept of `stackSize` and `capacity` are implemented. The interface

[5] https://github.com/OpenJML/OpenJML/issues/757 (2021)
[6] tested with versions 0.8.58 and 0.16.0-alpha-8

only cares that these concepts exist and can use them for verification.

3.6 Verification with KeY

With the interface, implementation, and usage all verified with OpenJML, we can take a look if the code also verifies using KeY. When loading the code into KeY, we are first met with an error message that `@Override` cannot be found on the classpath. Although irritating, this can be resolved by removing those annotations from the implementation (or adding the annotation to KeY's classpath). After that, we get an error that KeY cannot create a locset from `stackSize`, relating to the `assignable` clause on `pop`. Without diving deeper into why this happens, it can be worked-around by changing the clause to `\everything`. Finally, we get some warnings that in-group clauses are not supported by KeY. However, the code still loads successfully and the contracts can all be proven successfully.

Unfortunately, KeY does not recognize the `assert` statements in the usage, so this part cannot be verified easily. This is apparently a known issue, where the `assert` statement must be followed by a non-empty code block. When artificially adding such code blocks, the solver gets stuck on the `assert` statements, unable to automatically verify them. It is not obvious why KeY has trouble verifying the `assert` statements, but delving into interactive solving approaches is out of scope for this paper. Especially, since a focus of this paper is on ease-of-use for developers and students new to the field of formal verification, who likely want to avoid manually intervening in the proof attempt.

4 Related Work

The academic field of formal verification is quite broad. This section presents some recent studies relating to Java. Brizhinev published a case study similar to this paper, comparing different tools for formal verification of Java programs [BG18], but without a focus on teaching.

4.1 Proving the JDK

Two papers have analyzed the two different sorting algorithms implemented in the JDK: TimSort [GRB+15] and QuickSort [BSSU17]. Notably, de Gouw et al. have discovered a bug in the implementation during their attempt at proving TimSort. Both of these studies used KeY, showing that the tool is suitable, and possibly prefered, for proving complex algorithms.

4.2 Teaching Formal Verification with Java

Divasón and Romero published a paper on using Krakatoa as part of their course on formal verification [DR19]. That course is compulsory in the fourth semester of the bachelor's degree at the *University of La Rioja*. They concede as well that Krakatoa is notoriously hard to install. However, they worked around the issue by providing lab computers with Krakatoa pre-installed for their students.

The *Karlsruher Institute for Technology* uses KeY in their course on formal methods, targeted at both bachelor and master students [Bec16].

The *Chalmers University of Technology* uses both KeY and OpenJML in their course "Formal Methods for Software Development" [Ahr21]. For ease-of-use, they provide a web-interface to OpenJML to their students to get started[7].

4.3 Java Card

Java Card is a subset of the Java programming language, targeted at embedded systems [Zhi00]. The primary use-case of Java Card nowadays is in ATM cards and other smart cards. While Java Card is a subset of Java, it can still be considered a specialized language, since it was built for a very specific purpose. As a bonus, Java Card is sufficiently simplified that it becomes much more viable to verify large programs. In fact, one paper published a formally verified reference implementation of the Java Card API [Mos07]. This verification was conducted with KeY.

4.4 Java Pathfinder

Java Pathfinder is a tool for verifying the bytecode of Java programs, originally developed by the *NASA Ames Research Center* [HP99]. Java Pathfinder acts as a Java Virtual Machine (JVM) in that it loads and executes parts of the Java program's bytecode. However, instead of running the program normally, Java Pathfinder exploratively searches different execution paths of the program in an attempt to find inconsistent states, deadlocks, and other errors.

5 Conclusion

Although currently not well-known in mainstream development, there exist a variety of tools in the field of formal verification of Java programs. When it comes to formal specification of the Java source code, JML seems to be the undisputed standard. However, many tools add their own custom extensions to JML to cover specific cases or work around limitations. In that regard, the specification of a more complex program often results in tool-specific annotations, defeating the purpose of a general-purpose specification language. Nonetheless, pure JML is already very powerful and often sufficient for simpler code.

5.1 Mainstream Java Development

In mainstream development, ease-of-use and simplicity are paramount. Most software developers do not have the necessary knowledge or interest to deeply understand the theory behind the formal methods. For formal verification methods to find a foothold in mainstream development, writing specifications and verifying them should only be marginally

[7] http://cse-212294.cse.chalmers.se/courses/sefm/openjml/ (2021)

more time-intensive than writing automated tests. For those reasons, OpenJML looks most promising in finding potential for verifying, for example, important core libraries. However, the general effort of verifying code still seems too large for it to be worth to attempt to verify entire business applications.

5.2 Expert Usage

KeY seems to be the tool of choice amongst experts. This is supported by multiple papers using KeY when verifying complex algorithms [GRB+15][BSSU17][Mos07]. If a Java library is mission critical, teams might have one or more expert verification engineers on board. In those cases, OpenJML probably loses its appeal, as it does not provide any interactivity and only limited insight into the proof attempt. With Krakatoa's development being frozen, KeY seems like the natural choice.

5.3 Cutting-Edge Development

For developers dabbling in the newest and freshest features of Java, OpenJML comes closest to being useful. Although no tool currently supports the newest features, OpenJML is the only tool being re-written for the Java 16 platform. This is promising in that new features could likely be integrated sooner than for other tools still relying on older versions.

5.4 Teaching Formal Verification

For a quick introduction into formal verification, OpenJML provides the smallest barrier-of-entry. Students can start verifying code in a matter of seconds. When diving deeper into the theory, both KeY and Krakatoa provide insight into the inner workings of the theorem solver. According to Divasón and Romero, Krakatoa is the superior tool in terms of visualizing and explaining what is happening [DR19]. However, it might be adivsable to wait until development on Krakatoa has picked back up and is integrated into the new Why3 platform. While Krakatoa is very hard to install, a pre-installed environment provided to students could be a useful workaround.

5.5 Limitations of Formal Verification

As with many things, formal verification processes are only as strong as their weakest link. In mathematics, if something is proven, we often have 100% confidence in its correctness. However, if a piece of code passes a formal verifier, it would be naive to believe it is 100% correct. First of all, a formal proof can only ever be as accurate as the specification. As specifications are usually written by humans, there is always a potential for errors or inaccuracies there.

Additionally, there is the possibility for bugs in the verification software itself. While writing this paper, a bug in OpenJML was found, where OpenJML successfully verified code, which obviously did not satisfy the specification[8]. Fortunately, the bug was fixed within only a couple of days. However, such incidents show that false negatives in formal verification software is possible, and likely present in some form or another.

Lastly, modern software runtime environments are highly complex. Often virtualized, there are dozens or even hundreds of both software and hardware components involved in executing some program. For confidence in the correct execution of the program to be as high as possible, every single component involved would need to be formally verified. This is often unfeasible, or completely out of the developer's control. Even if every single software and hardware component was rigorously formally verified, there are other environmental factors, such as cosmic rays, which are impossible to predict [ZL79].

Nonetheless, by applying formal verification techniques, we can significantly improve the confidence in the most important parts of our code. In a landscape of ever-increasing complexity, it is even more important that as many components as possible are of high quality. Bringing the tools and methods of formal verification closer to students and developers is an important step in that direction.

References

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Matthias Ulbrich. *Deductive Software Verification – The KeY Book*. Springer International Publishing, 2016.

[Ahr21] Wolfgang Ahrendt. Chalmers - Formal Methods for Software Development. https://web.archive.org/web/20211122113236/https://wolfgangahrendt.gitlab.io/FMSD/, 2021. Accessed: 2021-11-22.

[Bec16] Bernhard Beckert. KIT - Wintersemester 2016/2017 - Formale Systeme. https://web.archive.org/web/20211122112348/https://formal.kastel.kit.edu/teaching/FormSysWS1617/, 2016. Accessed: 2021-11-22.

[BG18] Dmitry Brizhinev and Rajeev Goré. A case study in formal verification of a Java program. *ArXiv*, abs/1809.03162, 2018.

[BSSU17] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. Proving JDK's Dual Pivot Quicksort Correct. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE 2017)*, July 2017.

[Cok14] David R. Cok. OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*, volume 149 of *EPTCS*, pages 79–92, 2014.

[Cok21] David R. Cok. JML and OpenJML for Java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 65–67, New York, NY, USA, 2021. Association for Computing Machinery.

[DR19] Jose Divasón and Ana Romero. Using Krakatoa for Teaching Formal Verification of Java Programs. In Brijesh Dongol, Luigia Petre, and Graeme Smith, editors, *Formal Methods Teaching*, pages 37–51, Cham, 2019. Springer International Publishing.

[8] https://github.com/OpenJML/OpenJML/issues/754 (2021)

[GRB+15] S. Gouw, Jurriaan Rot, Frank Boer, Richard Bubel, and Reiner Hähnle. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In *International Conference on Computer Aided Verification*, pages 273–289, 07 2015.

[HMS98] Reiner Hähnle, Wolfram Menzel, and Peter H. Schmitt. Integrierter deduktiver software-entwurf. *Künstliche Intell.*, 12(4):40–41, 1998.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969.

[HP99] Klaus Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2, 10 1999.

[Lab15] Laboratoire de Récherche en Informatique. Krakatoa and jessie: verification tools for java and c programs. https://web.archive.org/web/20211028120112/http://krakatoa.lri.fr/, 2015. Accessed: 2021-10-08.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML: A Notation for Detailed Design*, pages 175–188. Springer US, Boston, MA, 1999.

[Lê11] Nhat Minh Lê. Contracts for java : A practical framework for contract programming. 2011.

[Mos07] Wojciech Mostowski. Fully Verified Java Card API Reference Implementation. *Water Science and Technology - WATER SCI TECHNOL*, 259, 01 2007.

[MSW+21] Steven Monteiro, Erikas Sokolovas, Ellen Wittingen, Tom van Dijk, and Marieke Huisman. IntelliJML: A JML Plugin for IntelliJ IDEA. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, page 39–42, New York, NY, USA, 2021. Association for Computing Machinery.

[Zhi00] Chen Zhiqun. *Java Card™ Technology for Smart Cards*. Addison-Wesley, 06 2000.

[ZL79] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, 1979.

A IntStack with stack size tracking

```
interface IntStack {

    /*@ public model instance int stackSize;
      @ public model instance int capacity;
     */

    /*@ public instance invariant stackSize >= 0;
      @ public instance invariant stackSize <=
           capacity;
     */

    /*@ requires stackSize < capacity;
      @ assignable \everything;
      @ ensures stackSize == \old(stackSize) + 1;
      @ ensures capacity == \old(capacity);
      @ ensures top() == value;
     */
    void push(int value);

    /*@ requires stackSize > 0;
      @ assignable stackSize;
      @ ensures stackSize == \old(stackSize) - 1;
      @ ensures \old(top()) == \result;
     */
    int pop();

    /*@ requires !isEmpty();
      @ pure
     */
    int top();

    /*@ ensures \result == (stackSize == 0);
      @ pure
     */
    boolean isEmpty();

    /*@ ensures \result == (stackSize == capacity);
      @ pure
     */
    boolean isFull();
}
```

Listing 17. Final version of `IntStack`

```
class IntStackArrayImpl implements IntStack {
    /*@ private invariant topIndex < values.length;
      @ private invariant topIndex >= -1;
     */

    private final int[] values; //@ in capacity;
    private int topIndex = -1; //@ in stackSize;

    /*@ private represents stackSize = topIndex + 1;
      @ private represents capacity = values.length;
    */

    /*@ requires size >= 0;
      @ ensures stackSize == 0;
      @ ensures capacity == size;
     */
    public IntStackArrayImpl(int size) {
        values = new int[size];
    }

    @Override
    public void push(int value) {
        values[++topIndex] = value;
    }

    @Override
    public int pop() {
        return values[topIndex--];
    }

    /*@ also
      @   private normal_behavior
      @     requires !isEmpty();
      @     ensures \result == values[topIndex];
     */
    @Override
    public int top() {
        return values[topIndex];
    }

    @Override
    public boolean isEmpty() {
        return topIndex == -1;
    }

    @Override
    public boolean isFull() {
        return topIndex == values.length - 1;
    }
}
```

Listing 18. Final version of `IntStackArrayImpl`

```
class IntStackUsage {

    public static void main(String[] args) {
        IntStack stack = new IntStackArrayImpl(2);
```

```
5          //@ assert stack.capacity == 2;
6          //@ assert stack.stackSize == 0;
7
8          stack.push(1);
9          //@ assert stack.stackSize == 1;
10         //@ assert stack.top() == 1;
11
12         stack.push(2);
13         //@ assert stack.stackSize == 2;
14         //@ assert stack.top() == 2;
15
16         int a = stack.pop();
17         //@ assert stack.stackSize == 1;
18         //@ assert a == 2;
19
20         stack.push(3);
21         //@ assert stack.stackSize == 2;
22         //@ assert stack.top() == 3;
23
24         int b = stack.pop();
25         //@ assert stack.stackSize == 1;
26         //@ assert b == 3;
27
28         stack.pop();
29         //@ assert stack.stackSize == 0;
30         //@ assert stack.isEmpty();
31     }
32 }
```

Listing 19. Final version of `IntStackUsage`

B Dockerfile for running OpenJML

```
1  FROM ubuntu:20.04
2
3  RUN apt-get update \
4   && apt-get install --assume-yes --quiet \
5    libgomp1 \
6    unzip \
7    wget \
8   && rm -rf /var/lib/apt/lists
9
10 ARG OPENJML_VERSION=0.16.0-alpha-8
11
12 RUN wget https://github.com/OpenJML/OpenJML/releases
       /download/${OPENJML_VERSION}/openjml-ubuntu-
       latest-${OPENJML_VERSION}.zip -O /tmp/openjml.
       zip
13
14 RUN unzip /tmp/openjml.zip -d /var/lib/openjml \
15  && ln -s /var/lib/openjml/openjml /usr/bin/openjml
        \
16  && chmod +x /usr/bin/openjml /var/lib/openjml/
        openjml
17
18 ENTRYPOINT [ "openjml" ]
```

Listing 20. Dockerfile for running OpenJML

```
1  $> docker build \
2      --file openjml.dockerfile \
3      --build-arg OPENJML_VERSION=0.16.0-alpha-8 \
4      --tag openjml:0.16.0-alpha-8 \
5      .
```

Listing 21. Building the OpenJML Docker image

```
1  $> docker run --rm \
2      -v "/path/to/local/sources:/opt/ws" \
3      openjml:0.16.0-alpha-8 -esc \
4      -sourcepath /opt/ws/src/main/java \
5      -verboseness 2 \
6      /opt/ws/src/main/java/my/package/MyFile.java
```

Listing 22. Example usage of OpenJML Docker image