Reasoning about JML: Differences between KeY and OpenJML

Jan Boerman, Marieke Huisman, and Sebastiaan Joosten

University of Twente, The Netherlands
j.g.j.boerman@student.utwente.nl,
{m.huisman,s.j.c.joosten}@utwente.nl

Abstract. To increase the impact and capabilities of formal verification, it should be possible to apply different verification techniques on the same specification. However, this can only be achieved if verification tools agree on the syntax and underlying semantics of the specification language and unfortunately, in practice, this is often not the case.
In this paper, we concentrate on one particular example, namely Java programs annotated with JML, and we present a case study in understanding differences in the treatment of these specifications. Concretely, we take a collection of JML-annotated programs, that we tried to reverify using KeY and OpenJML. This effort led to a list of syntactical and semantical differences in the JML support between KeY and OpenJML. We discuss these differences, and then derive some general principles on how to improve interoperability between verification tools, based on the experiences from this case study.

Keywords: Java Modeling Language · Static Verification · OpenJML · KeY.

1 Introduction

As a society, we increasingly rely on digital technology driven by software, and therefore we need formal techniques to provide guarantees about the quality and reliability of software. There is a wide plethora of tools and techniques available that contribute to this. However, all these tools and techniques have their own strong and weak points, and in order to increase impact and usability of formal techniques, we will need to find ways to combine them.

Unfortunately, combining tools is often not that straightforward, because even though in principle they implement the same specification language, they differ in the details, both in the syntax and the semantics.

This paper presents a case study to investigate the chances and difficulties for tool interoperability in one specific setting, namely Java programs annotated with JML (the Java Modeling Language) [18]. A wide range of different tools exist that take JML-annotated Java programs as input, and implement checks to establish whether the program behaves as specified by the annotations. Moreover, there are even tools that try to automatically come up with a suitable

JML specification for a given Java program, see e.g. [5] for an overview of JML tools. Despite efforts to agree on a common core language for JML, in practice there are still a lot of differences between all these tools, both syntactically and semantically. To improve this situation it is important to obtain a precise understanding of these differences.

In this paper, we consider two tools that take JML-annotated Java programs as input, and then apply static verification support, namely KeY [1] and OpenJML [9]. KeY is an interactive program verifier, based on dynamic logic, which typically makes it suitable for the verification of complex methods, because the user can incrementally build the proof. In constrast, OpenJML works fully automatically: from the annotated program, verification conditions are generated and sent to a first-order prover. This makes verification very fast for typical boilerplate methods (getters and setters) where the correct specifications can be given directly, but is less suited for incremental development of a specification. Thus, there is a high potential to increase verification efficiency if a user can smoothly switch between OpenJML and KeY during the verification process.

In order to investigate whether this switching could indeed be a smooth process, we took several sources of annotated examples that we tried to reverify in KeY and OpenJML. These include the examples from the KeY website (from www.key-project.org), as well as some hand-crafted examples that came up in the investigation of which keywords are supported and which are not. This resulted in a list of syntactical and semantical differences between the tools that should be addressed, or at least made explicit, so a user knows where to expect the differences. For all these examples of interest, we developed a minimal variant of the program in order to illustrate the issue in isolation. We do not believe this list of differences to be exhaustive, however we believe that it nicely illustrates the typical issues that have to be understood in order to enable tool interoperability.

It is important to stress that with this paper, we do not wish to argue for one tool over the other; we only would like to make it clear that there are differences in their behaviour, which can be unexpected for a tool user. We hope that the comparison helps eventually to make it easier to switch between different verification tools. Importantly, the authors of this paper have not been involved in the development of OpenJML or KeY, but they have a thorough experience with JML annotations, and are teaching JML-style specifications to Bachelor and Master students. Therefore, even though we try to find reasons to explain the differences in behaviour of KeY and OpenJML, the real reasons might be different. And of course, the differences that we identify between KeY and OpenJML are not just a warning for users of these tools: they are also an invitation to developers of these tools and developers of the JML standard to come to an agreement on a unified semantics for JML.

Finally, the last part of this paper tries to derive some general lessons from the experiences obtained with this case study in tool comparison. How can we avoid having to do such tool comparisons between all possible combinations of verification tools? How can we improve the situation upfront, in such a way that the potential differences between tools become more apparent? Or should we simply aim for a situation where all differences between tools are resolved? We hope that these lessons learned will inspire the community to look more closely at tool interoperability and the necessary steps to achieve this.

Contributions The main contribution of this case study is that it presents a collection of small JML-annotated programs for which verification behaves differently in KeY and OpenJML. Each program is designed to be minimal, and to illustrate a single issue in isolation, which makes them easy to understand and analyze for users and developers of tools and the JML language.

These examples should help users of KeY and OpenJML to better understand how easy or difficult it will be to switch between the two tools. They should also help KeY and OpenJML developers to better understand the strengths and weaknesses of their own tool, in comparison to the other tool. Finally, they should help developers of the JML standard to know what parts of the standard need to be clarified and what needs to be accounted for.

Scope of the examples Even though this paper may not cover all differences between KeY and OpenJML, we tried to choose our example set in a systematic manner. First, we systematically went through the documentation of JML [18] and tried to verify 'firsttouch' features individually. Second, we took the examples from the KeY website, and tried to verify these both in KeY and OpenJML. Third, we looked at the self-reported differences from the JML standard [12, 13], to come up with examples there. Finally, we created our own small examples in an ad-hoc fashion if a suspicion arose of a possible difference, while working with the examples stated above.

Related Work There are two webpages that list differences between JML and OpenJML [13], and differences between JML and KeY [12], respectively. However, these webpages focus on keyword support only – we will discuss in Section 3 – and are very brief.

Another comparison is made in the appendix of the JML standard itself [18, Section D], namely between JML and the specification language of ESC/Java (Extended Static Checker for Java) [19], which is a Java annotation language very similar to JML. Unfortunately, this comparison is outdated: it seems not to have been updated since 2003. And in particular, it does not contain a comparison between JML and the successors of ESC/Java, ESC/Java2, and later OpenJML. Fortunately, such a comparison is made upon the introduction of ESC/Java2 by the authors themselves [8].

We are not aware of many similar comparisons between tools. One of the authors of this paper has made a comparison between the interactive theorem provers PVS and Isabelle/HOL [10]. A comparison between several tools (including KeY and OpenJML) is made by Thüm et al. [21] for the purpose of aggregating the tools into a single model checking and theorem proving, focusing mainly on similarities. What is new in this case study is that we take a collection of externally developed examples, and base our experiences on this.

After reading an earlier version of this paper, David Cok, the main author of OpenJML, has given us some feedback through private communication. Where relevant, we included his remarks (clearly indicated as being his).

Overview of this Paper The remainder of this paper is organised as follows. Section 2 gives a short introduction to JML and briefly describes OpenJML and KeY. Section 3 then discusses several syntactical differences between JML as supported by KeY and by OpenJML. Then, Section 4 continues with the semantical differences we have observed. Finally, Section 5 draws some general lessons from our experiences.

2 Background: Static Verification with KeY and OpenJML

One way to verify correctness of Java code is by adding contracts to methods, and verifying those contracts. The standard in which to describe contracts for Java is JML [18, 17, 16]. A JML method contract essentially consists of two parts: one is called 'requires' and indicates the assumptions under which a method is called; the other is called 'ensures' and indicates the guarantees that the method gives. A small example is given in Figure 1. This specification states that the method should only be called with arguments `d` and `v` being strictly positive, and that it will return a value in the interval `[v - d, v]`. JML allows several variants on this: depending on whether or not a method is expected to always terminate, and whether or not it could throw exceptions, different kinds of contracts can be chosen.

Verifying JML contracts can be done in several ways. Popular approaches include runtime verification and static verification. Runtime verification will check validity of contracts during the execution of the program, while static verification will try to establish statically, without executing the program, whether the contracts are always respected. Runtime and static verification are orthogonal approaches: runtime verification finds errors when they happen, but does not verify the correctness of all possible programs, while static verification aims to prove correctness of all executions, but might indicate errors that will never happen during an execution. In particular, if a program does not have sufficient annotations, static verification might fail, even though the program is correct. However, if there is an issue with the program, it will be reported.

In static verification, typically only the contract of a method is used to reason about invocations of that method: The 'requires' part of a contract is the precondition, to be proven in the state before a method is called. Then the 'ensures' part of that contract, which is its postcondition, can be assumed in the state after its call. This makes static verification with JML highly modular: implementations can be changed freely as long as the contract remains provable, and the rest of the verification effort will remain valid. For a detailed analysis on the benefits of using contracts for verification (over inlining), see the work by Knüppel et al. [15], which includes experiments in KeY.

```
1 class SimpleContract {
2
3     /*@ requires d > 0 && v > 0;
4       @ ensures v - d <= \result && \result <= v;
5       @*/
6     public static int round(int v, int d) {
7       return v - (v % d);
8     }
9 }
```

Fig. 1. Method with a contract

A static verification tool transforms the specified program and its contracts into proof obligations. How the translation is done depends on the tool used. These proof obligations are then checked in some way, which again depends on the verification tool used. Not only the contract specification is translated during such a transformation, but certain implicit language rules are as well. In Figure 1, there is a potential division-by-zero-error on line 7. However, the precondition of the method suffices to show that this division-by-zero-error will not occur.

OpenJML OpenJML [9] is developed as the successor of ESC/Java2 and the runtime verification tool suite for JML [7]. For static verification, it transforms a JML-annotated program into a static single assignment form, and then generates first-order logic verification conditions from this transformed program. This output format is suitable for a satisfiability modulo theory (SMT) solver. As such, it is given as input to an SMT solver, which by default is Z3 [20].

OpenJML can be used by invoking it from the command line, or as an Eclipse plugin. OpenJML can do run time verification as well as static verification. Throughout this case study, we invoke OpenJML from the command line with `-esc` plus a file name, to do static verification. The result can be a set of warnings and errors, or if the program has no issues, nothing at all. Indeed, for the program in Figure 1, no output is given, indicating the program is correct. If verification does not go through, the warning or error points to the places in the program that are causing the issue. For instance, changing line 7 to **`return`** `d;` will give a postcondition violation error that points to line 4, as well as to line 7.

David Cok let us know that OpenJML also has an IDE in which counterexamples can be explored. We did not try the IDE for this work.

KeY The KeY project positions itself as a portfolio of tools for program verification [1]. It has a tool for static program verification, allows test generation based on contracts, and has an Eclipse plugin for symbolic debugging. All of these tools are built on a common code base that includes an interactive prover and a symbolic evaluator of programs. KeY is based on dynamic logic, shared by the symbolic evaluation and the interactive prover, into which programs and annotations are translated.

KeY allows additional axioms and strategies to be added to its prover through a feature called 'add user defined taclets'. Even the assumptions made in KeY's built-in taclets can be tweaked: there is a setting called 'taclet options' that allow us to determine how certain Java commands are treated. For instance, one can tell KeY to ignore assert commands to mimic the behavior of when the Java Virtual Machine (JVM) is called in the same way. Alternatively, one can tell KeY to treat failing assert commands as runtime exceptions, or to generate proof requirements that ensure that they hold. Proving that assertions hold means that the JVM will not report any assertion failures, regardless of whether assertion checking is enabled in the JVM. KeY allows us to save proofs, both completed and incomplete proofs. For this case study, we refrained from using any user defined taclets.

In the tool KeY, one opens a directory from a GUI. The tool then lists all proof obligations for all JML annotated files in that directory. When one is selected, one can interactively create a proof for it. There is also an automatic option. Using the automatic option on the program in Figure 1 produces a proof with two open subgoals: One where a proof of `jmod(v, d) <= d` is required, and another where `jmod(v, d) < 0` is assumed, where `jmod` refers to the built-in function of the JVM that computes the modulo, which arises as a translation of `%`. KeY makes no assumptions about the JVM's implementation of `jmod`, so we cannot complete the proof. This does not depend on modes of taclets (like whether or not arithmetic is verified with overflow checks). We could, however, proceed by adding these assumptions to KeY manually by adding taclets, and in this way complete the proof.

3 Syntactical Differences

This section describes differences between KeY and OpenJML for which a syntactic criterion can be given. We first discuss differences in the parts of the JML standard that are covered by OpenJML and KeY. Then, we continue with the extensions to the JML standard offered by either KeY or OpenJML. Finally, we also briefly discuss what could be done to decrease the syntactical gaps. Table 1 summarizes the discussion in this section, and gives an overview which keywords are supported by which tool (and by the JML standards). This table does not show what is supported by the parser built into each tool, but rather by the tool when performing static verification. We have also omitted keywords for which we could not find clear differences, such as **`\forall`**.

Covered JML Subset For most syntactical differences, the JML standard seems to be a driving force: both KeY and OpenJML implementers aim to let JML keywords behave as described in the JML manual. The developers are generally aware of the syntactical differences: both the KeY website and the OpenJML website feature lists of differences and similarities between the tool and the JML

Table 1. Syntactic elements evaluated, and support in JML, OpenJML and KeY

Keywords	JML	KeY	OpenJML
`\sum \product \num_of`	Yes	Yes	No
`\strictly_nothing strictly_pure`	No	Yes	No
`\not_assigned`	Yes	No	No
`\bSum \bProduct`	No	Yes	No
`\locset \intersect \set_union`	No	Yes	No
`\distinct`	No	No	Yes
`\index`	No	Yes	Yes
Certain Java arithmetic: `%` `^`	Via Java	No	Yes

standard [12, 13]. Some JML keywords that KeY supports, but OpenJML does not are: **\sum**, **\product** and **\num_of**.

Non-JML Extensions Other syntactic differences come from non-JML extensions to KeY. KeY has greater flexibility than OpenJML in indicating that certain variables may not be assigned: in OpenJML, like KeY, one can state for which variables the value may change through the execution of the method. All other variables should never be assigned. In KeY, however, one by default states that other variables will eventually return to their original value. The difference can be useful in concurrent programs (although neither KeY nor OpenJML currently support the verification of such programs). In KeY, within an **assignable** clause, the **\strictly_nothing** keyword indicates that no global variables may be assigned, even if they are eventually restored to their old value. Similarly, **strictly_pure** indicates the same thing. These keywords are not part of the JML standard, they implement what according to the JML standard should be the behaviour of **\nothing** and **pure** respectively. OpenJML does not offer a choice on how **assignable** is interpreted, but it does implement the JML standard by default. The JML keyword **\not_assigned**, which also serves this purpose in the JML standard, is not supported by OpenJML or KeY.

In conditions, the keywords **\bSum** and **\bProduct** are used as a bounded verifier-friendly version of **\sum** and **\product**, respectively, which gets the range over which the sum or product is calculated as two integers.

For modeling the heap, KeY introduces the **\locset** type, as well as set operations like **\intersect**, **\set_minus** and **\set_union**. Reasoning about the heap was introduced to support dynamic framing in KeY [22, 2].

Like KeY, OpenJML has some non-JML extensions that are not supported by KeY. For conveniently writing that every pair in a set of variables is distinct, OpenJML writes **\distinct**, which can be considerably shorter than using $!=$ pairwise for large sets of variables.

Interestingly, the keyword **\index** is supported in both OpenJML and KeY, even though this keyword is not part of the JML standard. Within an enhanced for-loop (a for-each loop), **\index** is used to indicate the current index. The

`\index` keyword was discussed at a JML workshop, and may become part of JML as the `\count` keyword.

We did not focus on finding out which subset of Java is supported by KeY or OpenJML. However, as mentioned in the introduction, Figure 1 did not automatically verify in KeY while it did verify in OpenJML. This seems to happen because KeY does not have a built-in axiomatization of `%` by default. Similarly, KeY could not verify any properties about the bit-wise xor, or ^ in Java, while OpenJML could. In contrast to OpenJML, KeY does not allow the use of Java generics, although the KeY website claims that these can be removed statically via an Eclipse plugin (which we did not test).

Reducing the Gap The syntactical differences between KeY and OpenJML can be a nuisance for someone trying to use different tools for different parts of the same specification, and therefore these differences should be clearly documented and avoided as much as possible.

We recommend that KeY supports the `\not_assigned` keyword, and deprecates `\strictly_nothing` and **strictly_pure**. With the exception of `\locset` and the corresponding set operations, all keywords can be expressed in standard JML. For the use of `\locset`, it is worth considering adding this to the JML standard. We also recommend to add `\index` to the JML standard.

At some point, there has been a proposal to add markers to annotations, to indicate that they were tool-specific, because KeY would require different in-code annotations than e.g. OpenJML. One could imagine that annotations that use `/*KEY@ ....*/` as surrounding comments are considered only by KeY. This idea was introduced during a JML workshop, and is now supported by at least OpenJML through the markers `RAC`, `ESC`, and `OPENJML`.

4 Semantical Differences

The previous section discussed syntactical differences between the JML specifications supported by OpenJML and KeY. However, even more important are semantical differences, where the specifications are the same (maybe modulo syntactical differences), but the behaviour of the tools is different.

This section provides a list of such differences. We do not believe that this list is exhaustive, but we believe it gives a good impression of the semantical differences in tool behaviour that one should be aware of. In fact, we are not sure whether it is possible to give a fully exhaustive list of such differences, but we believe that understanding and discussing the differences is important for a better interoperability between tools.

The sources of the differences that we list here can vary: sometimes they are caused by the underlying prover technology (or might even be caused by a bug in the underlying solver), but they can also be related to a different semantical interpretation of the Java or JML semantics.

Compiler Checks A difference that stands out immediately is that KeY does not require a class to be compilable, while OpenJML does require this. David Cok told us this is due to OpenJML's use of OpenJDK to produce ASTs.

The KeY approach provides flexibility, and has several advantages:

– it makes it possible to verify classes in isolation, without considering the complete hierarchy of all classes surrounding this class, and
– it is possibly to quickly copy the class that is being verified into a different file, without having to change the class name accordingly.

However, the disadvantage and major risk is that one might spend a lot of time on the verification of a non-compilable program (and this time might thus be completely wasted). The KeY approach thus requires more discipline from the users to make sure that they are indeed working on a correct Java file.

In contrast, OpenJML builds this check in, and thus immediately identifies program errors, but does not make it easy to verify single classes in isolation. As a result, in OpenJML one often has to spend a lot of time on stripping irrelevant imports, function calls etc., in order to make the tool actually check some specification.

Visibility Checks Related is that KeY does not do visibility checks on fields and methods. In particular, it does not do these checks in the specifications. As a result, KeY does not complain if a publicly visible specification uses private variables. For example, in Figure 2 the KeY-verified example uses private fields in the public method specification: both the private variable a and the private method ReturnFive occur in the **ensures** statement. In contrast, OpenJML immediately reports all visibility issues in this specification.

```
public class InitPrivateToPublic {
  private int a;
  /*@ ensures a == returnFive();
    @*/
  public InitPrivateToPublic() {
     a = returnFive();
  }

  /*@ ensures \result == 5; @*/
  private /*@ pure @*/ int returnFive(){
    return 5;
  }
}
```

Fig. 2. Publicly visible specification with private variables

This lack of visibility checking in KeY is in violation with the JML standard [18, section 2.4], and we believe that this is an omission in the KeY implementation, because it breaks the standard rules of encapsulating an object's internal state. Moreover, a simple solution is available by declaring the variable **spec_public**, which implicitly declares a model variable that abstracts from this internal state (and thus, if the internal state is changed, only the relation between the model variable and the internal state has to be adapted, but the public method specifications do not change).

```
public class InitPublic {
   private /*@ spec_public @*/ int a;
   /*@ public normal_behavior
     @ ensures a == 5;
     @*/
   public InitPublic() {
      a = returnFive();
   }

   private /*@ pure @*/ int returnFive(){
      return 5;
   }
}
```

Fig. 3. Method without a contract

Inlining KeY and OpenJML have a different approach to handling method calls. OpenJML uses a very puristic approach: any method call will be abstracted by its method specification. Thus, consider the example in Figure 3. Method `InitPublic` calls method `ReturnFive`. As method `ReturnFive` does not have any method specification, OpenJML will simply assume that any behaviour of this call is possible, and it will not be able to prove the postcondition `a == 5` (even though we can clearly see that the implementation of method `ReturnFive` achieves exactly this).

KeY follows a different approach here. If no postcondition is specified, i.e., no **ensures** clause is present, KeY will inline this method call, and thus the postcondition of method `InitPublic` can be proven. Notice that when a postcondition of `ReturnFive` is specified, even when this is only **ensures true;**, inlining will not happen anymore, and verification of method `InitPublic` will fail (except of course if the postcondition of `ReturnFive` captures that it returns 5).

KeY thus requires a user to think carefully about whether a method call will indeed always end up invoking the same method invocation. If the method `ReturnFive` may be overwritten, the postcondition of method `InitPublic` might not hold anymore after the call to `ReturnFive` in the subclass. Thus, again KeY provides extra flexibility, i.e., not requiring every call to be annotated, but at the risk of verifying something that is not correct. The JML semantics does not explicitly describe whether unfolding is a valid proof step for static verification.

The use of contracts, rather than inlining, can greatly speed up time required for running the automated verification, as witnessed by experiments in KeY done by Knüppel et al. [15]. Obviously, using inlining for program verification can avoid the need of even having to write contracts.

David Cok brought to our attention that OpenJML has some undocumented support for inlining. He suggests the introduction of an inline keyword to steer the desired behavior.

Memory Safety and Exceptional Behaviour An interesting difference that we noted between KeY and OpenJML is in the checks that are implicitly added to ensure memory safety. The JML semantics advocates a non-null by default semantics, but it does not exclude other exceptions by default.

KeY has three ways of dealing with exceptions, depending on a taclet settings. One setting requires you to prove that no kind of exception will ever be thrown. This does not allow you to verify any program for which throwing errors is part of the specification. Another setting assumes no exceptions occur, making the analysis unsound, but possibly still useful for catching certain kinds of bugs. A final setting, and the one used in the discussion below, is to treat all exceptions as exceptional behavior.

Consider for example the small fragment in Figure 4. This specification expresses that it will throw an `ArrayIndexOutOfBoundsException`. KeY verifies this example without any problem, but OpenJML does not. Instead, it complains that for the expression `a[-1]` it cannot verify that the index `-1` is within the bounds of the array. Thus: OpenJML adds implicit checks that ensure that this runtime exception will never be thrown, and does not allow the user to prove that this exception actually will be thrown here.

```java
public class C2 {
  /*@ private exceptional_behavior
    @ requires true; // is by default but we have to write something
    @ signals (ArrayIndexOutOfBoundsException) true;
    @*/
  public int getZero(int[] a) {
          return a[-1];
  }
}
```

Fig. 4. Method with an exception as its contract

As a consequence, implicitly OpenJML reduces the use of exceptional behaviour specifications only to explicit exceptions, and is more rigorous on runtime exceptions than is prescribed by the JML standard [18, Section 9.8].

```
public class C {
  /*@ private normal_behavior
    @ requires a[a.length]!=0;
    @*/
  public int getZero(int[] a) {
          return 0;
  }
}
```

Fig. 5. Method raising an exception in its contract

A related difference is that OpenJML checks for bounds within the preconditions of contracts, while KeY does not. Therefore, the example of Figure 5 results in a warning in OpenJML, while KeY proves its correctness. The JML standard states two things about errors in contracts: First, it states that statements in a contract are to be evaluated in order, such that an exception like this can be prevented by verifying that the index is within bounds (i.e. verifying that `a.length<a.length` for our example). Second, it states that a condition is valid if it evaluates to true (also referred to as 'strong semantics'). This means that if exceptions are thrown in the evaluation of a condition, that condition is false. Consequently, this requirement makes the contract trivially valid (see also [3, 6] for related discussions). Despite these two statements, we cannot deduce from the JML standard how static verification tools should deal with exceptions in preconditions. We argue that preconditions that evaluate as exceptions are always undesirable and point to errors, agreeing with Chalin [6] on this point. Therefore, we suggest OpenJML's way of dealing with this issue to become standard.

Initialisation Checks OpenJML and KeY also differ in the checks that they insert for variable initialisation. Consider the example in Figure 6. JML specifies that variables are always non-null by default, so the `getLength` function satisfies its contract. Therefore, we should reasonably deduce that the `InitArray` function contains an error.

This example is verified in KeY, but OpenJML complains. OpenJML reports that there is no explicit constructor, and mentions line 2 as problematic. Using the non-null default, the reference to array a should always be non-null, but this is not guaranteed in this program. Adding the initialisation `a = new int[0];` by uncommenting line 4 solves this issue.

We believe this difference is caused because KeY simply forgets to generate a proof goal for the initialisation of arrays, while OpenJML adds the implicit fact that a should be non-null as an implicit class invariant to the variable declaration, and therefore signals a problem.

Power of Underlying Solver In some cases, the capabilities of the underlying prover determine what can be verified. We previously stated that it can be

```
public class InitArray {
  private int[] a;
  InitArray(){
    // a = new int[0]; // missing
  }
  /*@ ensures \result >= 0; */
  public int getLength() {
    return a.length;
  }
}
```

Fig. 6. No array initialisation

worthwhile to combine different provers, depending on which prover is most suitable for the task. In doing this, we implicitly assume that differences exist in which annotated programs can be proven automatically, and which cannot. We show that this is indeed the case, even though this does not give us fundamental insights into how KeY and OpenJML interpret programs. We give three examples: one where both provers fail, a second where OpenJML is able to prove correctness automatically while KeY is not, and a third example where it is the other way around.

Consider first the code fragment in Figure 7.

This program, with the loop invariant as specified, could not be verified automatically by KeY: After letting KeY run automatically for an hour, still no solution was found. We believe a manual KeY proof exists, as this is claimed for a more elaborate version of this code [14].

When we try to verify this with OpenJML, verification fails within ten seconds. This makes the example one where both OpenJML and KeY fail to verify a program. Fortunately there was an easy fix, by splitting the big loop invariant into two separate loop invariants. That is: replace the `&&` on line 9 by `; loop_invariant`. This verified the program without any problem, again in roughly ten seconds. Notice that this is logically completely equivalent, but apparently using the full conjunction in the generated proof obligation is too complicated for the underlying first-order prover Z3. Thus, the OpenJML user has to be aware of this issue, and make sure that his or her specification style fits the capabilities of the underlying prover. If we run KeY on the changed program, it again fails to find a solution (in reasonable time).

For our second example, we did not manage to solve the issue. Consider the Least Common Prefix (LCP) program in Figure 8, which is part of a solution to a VerifyThis 2012 challenge [4, 11]. Verification of this algorithm works in KeY, but not OpenJML. David Cok pointed out that OpenJML can verify the example by replacing the maintaining clause on line 13 and 14 with:

```
(\forall int z; x <=z && z < x+l; a[z] == a[y+z-x])
```

```
public class ReverseArray_failing {
 /*@ public normal_behavior  diverges true; @*/
 public void reverse(int[] a) {
  int i = 0;
  final int length = (a.length/2) ;
  /*@
    @ loop_invariant (\forall int j; j>=0 && j<i;
                        \old(a[a.length-(j+1)])==a[j])
    @  && (\forall int j; j>=i && j<length;
    @         \old(a[a.length-(j+1)])==a[a.length-(j+1)] &&
    @         \old(a[j])==a[j]);
    @ loop_invariant i>=0 && i<=length;
    @*/
  while (i<length) {
    int tmp = a[a.length-(i+1)];
    a[a.length-(i+1)] = a[i];
    a[i] = tmp;
    i++;
  }
 }
}
```

Fig. 7. Complicated loop invariant

David Cok also points out that the issue is indeed with the underlying Z3 solver, which has trouble with quantified expressions that have arbitrary expressions as array indices.

5 Lessons Learned

This case study investigated differences among JML, OpenJML and KeY. Both tools aim to verify JML-annotated Java programs, but this case study shows that the differences in their behaviour are substantial. Therefore, at the moment it is a non-trivial exercise to reuse verified specifications from one tool by the other tool, even though the developers of OpenJML and KeY have had discussions to agree on a common semantics for a core of JML.

The differences fall in different categories: syntax of the specification language, interpretation of the JML semantics, behaviour of the underlying prover, and choice of defaults in programs and specifications. To improve interoperability between tools, we need to investigate if we can reduce these differences and if this is not possible, we should make sure that we document them. We believe that tool developers should take much more responsibility than they do currently to improve interoperability between tools. Looking at the different categories of differences that we identified, we believe the following should be aimed for:

```
/* @author bruns, woj */
final class LCP {

  /*@ normal_behavior
  @ requires 0 <= x && x < a.length;
  @ requires 0 <= y && y < a.length;
  @ requires x != y;
  @ pure @*/
  static int lcp(int[] a, int x, int y) {
    int l = 0;
    /*@ maintaining 0 <= l && l+x <= a.length
      @             && l+y <= a.length && x!=y;
      @ maintaining (\forall int z; 0 <= z && z < l;
      @                          a[x+z] == a[y+z] );
      @ decreasing a.length-l; @*/
    while (x + l < a.length && y + l < a.length
          && a[x + l] == a[y + l])
      l++;
    return l;
  }
}
```

Fig. 8. Modified LCP example

– Syntactical differences should simply be avoided. If tool developers feel the need to define their own syntax, we believe that they should provide users with a script to turn the annotated program into a standard-JML compliant version, or use special markers for the non-JML-compliant annotations.
– Differences in behaviour caused by the underlying prover should be avoided as much as possible. These are caused by the format in which proof obligations are sent to the underlying prover (and by the use of different underlying provers). Finding the optimal format is a research challenge, and it is important that tool developers exchange their experiences with this. The issue can probably also be further reduced by supporting different back-end provers.
– The differences due to a different interpretation of the JML semantics should be avoided as much as possible. Therefore, it is important to continue the discussion on a common semantics of core JML, and to document the outcome of this discussion. Also, tool developers should agree to adhere to the decisions made during this discussion, and if necessary, adapt their tool implementation. If a tool developer still decides to deviate from this common semantics, he or she should document this, or preferably provide a flag that allows one to still use the common semantics.
– For the differences caused by the choice of defaults in programs and specifications, the same applies: these should be documented, and an option could be provided as a special flag. In some cases, the tool might also decide to issue an explicit warning about the defaults chosen, and that other tools

might deviate from this. For example, it would help if the KeY tool would issue a warning that it did not check whether the Java program actually can be compiled.

And very importantly, these choices and assumptions that cause differences should be documented in a way that is understandable and accessible for people who did not develop KeY or OpenJML, as they are ones that are the most likely to benefit from tool interoperability. Ideally, tools should be developed with this idea of interoperability in mind. We understand that it might not be easy to change the complete implementation of a tool, but it would help users a lot if OpenJML could be invoked with a `-KeY` flag, and vice versa[1].

To improve the current situation, a first starting point would be to define a collection of verification benchmarks with intended behaviours (similar to the litmus tests for relaxed memory models). The examples discussed in this paper could be a starting point for this, but further extensions will be necessary.

In this case study, and also in the conclusions, we focused very much on JML-annotated programs. However, we believe that the general lessons learned also apply to other verification tools, and that it is time for the formal verification community to really put more effort in tool interoperability, in order to increase the impact of formal verification.

Acknowledgements The work of Huisman and Joosten is supported by NWO grant 639.023.710 for the Mercedes project.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification – The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing (2016)
2. Beckert, B., Klebanov, V., Weiß, B.: Dynamic logic for Java. In: Deductive Software Verification - The KeY Book - From Theory to Practice [1], pp. 49–106
3. Berg, J.v.d., Jacobs, B., Poll, E.: First steps in formalising JML. In: ECOOP workshop on Formal Techniques for Java Programs (FTfJP'2000) (2000)
4. Bruns, D., Mostowski, W., Ulbrich, M.: Implementation-level verification of algorithms with KeY. STTT **17**(6), 729–744 (2015)
5. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., Poll, E.: An overview of JML tools and applications. STTT **7**(3), 212–232 (2005)
6. Chalin, P.: Reassessing JML's logical foundation. In: Proceedings of the 7th Workshop on Formal Techniques for Java-like Programs (FTfJP'05), Glasgow, Scotland (2005)
7. Cheon, Y.: A Runtime Assertion Checker for the Java Modeling Language. Ph.D. thesis, Department of Computer Science, Iowa State University, Ames (2003), technical Report 03-09

[1] David Cok tells us that OpenJML can be invoked with a `-strict` flag, causing it to follow the JML semantics more strictly, which is already a great step.

8. Cok, D., Kiniry, J.R.: ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.L., Muntean, T. (eds.) Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004. LNCS, vol. 3362, pp. 108–128. Springer (2005)
9. Cok, D.R.: OpenJML: JML for Java 7 by extending OpenJDK. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 472–479. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
10. Griffioen, D., Huisman, M.: A comparison of PVS and Isabelle/HOL. In: Grundy, J., Newey, M. (eds.) Theorem Proving in Higher Order Logics. Lecture Notes in Computer Science, vol. 1479, pp. 123–142. Springer, Berlin, Heidelberg (1998)
11. Huisman, M., Klebanov, V., Monahan, R.: VerifyThis 2012 - A program verification competition. STTT **17**(6), 647–657 (2015). https://doi.org/10.1007/s10009-015-0396-8, https://doi.org/10.1007/s10009-015-0396-8
12. JML support in KeY, https://www.key-project.org/jml-support-in-key/
13. Currently supported features (April 2018), http://www.openjml.org/documentation/features.shtml
14. Kirsten, M.: Proving well-definedness of JML specifications with KeY. Master's thesis, ITI Schmitt, Karlsruhe Institute of Technology (November 2013)
15. Knüppel, A., Thüm, T., Padylla, C., Schaefer, I.: Scalability of deductive verification depends on method call treatment. In: 8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA). LNCS, Springer (2018), to appear
16. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems. pp. 175–188. Springer US, Boston, MA (1999)
17. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes **31**(3), 1–38 (2006)
18. Leavens, G., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D.R., Müller, P., Kiniry, J., Chalin, P.: JML Reference Manual (Feb 2007), dept. of Computer Science, Iowa State University. Available from http://www.jmlspecs.org
19. Leino, K.: Applications of Extended Static Checking. In: Cousot, P. (ed.) Static Analysis (SAS 2001). LNCS, vol. 2126, pp. 185–193. Springer (2001)
20. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
21. Thüm, T., Meinicke, J., Benduhn, F., Hentschel, M., von Rhein, A., Saake, G.: Potential synergies of theorem proving and model checking for software product lines. In: Proceedings of the 18th International Software Product Line Conference - Volume 1. pp. 177–186. SPLC '14, ACM (2014)
22. Weiß, B.: Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction. Ph.D. thesis, Karlsruhe Institute of Technology (2011)