Joining Forces! Reusing Contracts for Deductive Verifiers through Automatic Translation*

Lukas Armborst[0000−0001−7565−0954], Sophie Lathouwers[0000−0002−7544−447X], and Marieke Huisman[0000−0003−4467−072X]

University of Twente, Enschede, the Netherlands
{l.armborst,s.a.m.lathouwers,m.huisman}@utwente.nl

Abstract. Deductive verifiers can be used to prove the correctness of programs by specifying the program's intended behaviour using annotations such as pre- and postconditions. Unfortunately, most verifiers use their own unique specification language for those contract-based annotations. While many of them have similar concepts and syntax, there are numerous semantic differences and subtleties that make it very difficult to reuse specifications between verifiers. But reusing specifications could help overcome one of the bottlenecks of deductive verification, namely writing specifications. Therefore, we present the Specification Translator, a tool to automatically translate annotations for deductive verifiers. It currently supports Java programs annotated for OpenJML, Krakatoa and VerCors. Using the Specification Translator, we show that we can reuse 81% of the annotations, which would otherwise need to be manually translated. Moreover, it allows to reuse tools such as Daikon that generate annotations only in the syntax of one specific tool.

Keywords: Annotations · Specifications · Deductive verification · Translation · Tool interoperability.

1 Introduction

Deductive verification is a powerful technique that can be used to improve the reliability of software. It can be used to reason about e.g. memory safety and functional correctness, even if the system has an infinite state space and concurrency. There exist many deductive verifiers, some of which are better suited to certain problems than others. Therefore, it is important that tools cooperate such that users can select the tool best suited to their problem. For example, some deductive verifiers support the use of different solvers for individual proof tasks (e.g. Why3 [18], Krakatoa [17]). However, tool interoperability for deductive verifiers is much harder when it comes to reusing specifications, because most tools use their own unique specification language. As a result, it is currently impossible to switch between tools without investing significant time and effort to manually translate the specifications.

* This work was supported by the NWO VICI 639.023.710 Mercedes project.

Finding and writing the specifications is considered to be one of the large bottlenecks in applying deductive verification in practice [3,22]. This is why we would like to reuse specifications where possible, such as for APIs (e.g. [2,5,7,15,20,26]). Often the verification requires more lines of specification than lines of code. For example, one of the APIs [2] required four times as many lines of specification as lines of code. Unfortunately, with the current limited tool interoperability, it can be difficult for users to verify a program that uses one of these APIs. The user is either limited to using the same verifier for their program as was used for the library, or they need to spend significant effort to re-verify the library in their verifier of choice.

To enable the reuse of specifications, we propose the Specification Translator, a tool to automatically translate specifications between OpenJML [13], Krakatoa [17] and VerCors [6]. We specifically target deductive verifiers for Java programs. We chose to support OpenJML because it is one of the most well-known deductive verifiers for Java and it supports a large subset of Java Modeling Language (JML) [27]. Krakatoa has been included since it is no longer actively developed and thus it would be useful to port the specifications to another tool that is still being maintained. And, it is one of the few verifiers that supports using various solvers for individual proof tasks. VerCors has been included because it is based on separation logic, which showcases how to deal with extensions to standard JML. The Specification Translator makes it easier to share verified programs between these tools. Moreover, it can also be used to reuse results from other tools, such as specification generators like Daikon [16].

In this paper, we investigate to what extent specifications can be automatically translated to enable reuse between verifiers. To achieve this, we explore the semantic differences between verifiers. There are often commonalities between specification languages, as many Java-based deductive verifiers have a specification language inspired by JML. However, translating annotations is not as straightforward as it might seem since you need detailed knowledge about the specification languages and their semantic differences. For example, some of the things one needs to know when translating from tool X to Y, for example OpenJML to VerCors, include:

- Does tool X have any built-in assumptions? If so, do these correspond to the built-in assumptions of tool Y?
  - OpenJML assumes that variables are non-null by default whereas VerCors requires the user to write annotations to express this.
- Does tool Y support all the concepts used in the annotations for tool X? If not, does it have a similar concept?
  - OpenJML has `assignable` clauses to indicate whether you can write to a variable. VerCors does not have those, but it is built on permission-based separation logic and requires annotations indicating the amount of permissions for a specific memory location.
  - OpenJML supports `behavior` clauses, which can be used to make case distinctions in the specifications. VerCors does not support those, so one needs to rewrite pre- and postconditions with implications to indicate that something should hold only for a specific case.

– Even if Y supports a concept, does it have the same semantics as in tool X?
  • In OpenJML the term "predicate" refers to a boolean expression, whereas in VerCors it is a function that returns a boolean. As such, VerCors' predicates are more similar to model methods in OpenJML.

The Specification Translator handles most of these details automatically, reducing the manual effort for the user.

For the translation, the Specification Translator uses an intermediate representation which contains concepts supported by multiple tools, and makes many implicit assumptions explicit. This makes it easy to extend the Specification Translator with new input and output languages.

In our evaluation, we show that we can translate most annotations between the verifiers (81%), and we analyse whether the program can be verified after translation or how much effort is still required. Moreover, we show how we can use the Specification Translator combined with Daikon to generate specifications for VerCors. This used to be impossible without manual intervention.

Contributions In short, this paper introduces the tool Specification Translator, which can translate specifications between Krakatoa, OpenJML and VerCors. With this tool, we enable tool interoperability, and thereby prevent users from spending a lot of time re-doing existing work, such as library verification, and instead allow them to build on top of it and focus on new research instead. Our evaluation shows for 30 programs and 2 larger case studies that more than 80% of the specifications can be reused when using the Specification Translator, highlighting its effectiveness in reducing the effort of integrating tool results. The Specification Translator also supports the integration of tools like Daikon, which supports one single specification language for Java, with other verifiers, maximising their impact. Moreover, the paper highlights the differences in semantics between Krakatoa, OpenJML and VerCors, as well as how to translate between them.

Outline of the paper In the next section, we will describe the design of the Specification Translator. Section 3 explains some of the more intricate translations and design choices. Then, Section 4 evaluates the tool, specifically how many annotations can be reused and reverified. We discuss related work in Section 5 and conclude in Section 6.

2 Design of the Specification Translator

This section gives an overview of how the Specification Translator works (see Figure 1). We briefly explain which transformations are done on an example. Let us assume we have a file with OpenJML annotations that we want to translate into VerCors annotations. First, the file is converted into an OpenJML-specific syntax tree that resembles an *abstract syntax tree*. However, it is enriched with formatting information such as whitespace, which is normally left out of an AST. This allows all translations to remain as close as possible to the original

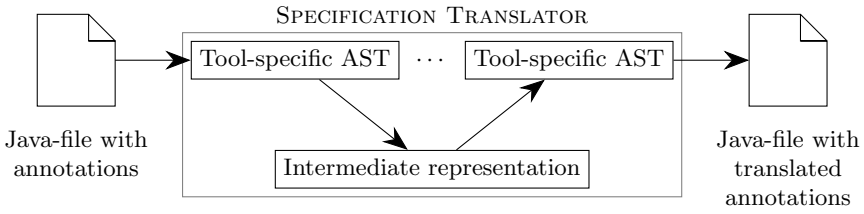

Fig. 1. Overview of how the Specification Translator works. Given an annotated Java-file, the input is first converted into a tool-specific AST. This is translated via a tool-independent intermediate representation into the tool-specific AST of the target tool. From there, the output file is generated.

file, including formatting. We will refer to these syntax trees as AST, despite the additional verbosity. Then, the OpenJML-specific AST is translated into the intermediate representation. OpenJML-specific annotations are commented if they are not supported by any other tool and thus cannot be translated. It will also desugar some expressions, such as making implicit assumptions explicit. Afterwards, the intermediate representation is translated into a VerCors-specific AST. This may again comment some annotations, if VerCors does not support them. Others may be rewritten, if they are not supported in the target specification language directly, but the intention can still be expressed. Using the VerCors-specific AST, we generate the output file with all translated annotations.

If there are annotations that are not translatable, the tool informs the user on the command line. In the file, instead of silently removing the problematic annotations, they are turned into comments. This ensures traceability, and helps the user in case manual intervention is needed.

Intermediate representation Given that many tools based their specification language on the Java Modeling Language (JML), the intermediate representation (IR) is also based on that standard. A few JML concepts were left out of the IR, most notably redundancy, such as `assignable_redundantly` or `example` definitions. This does not change the specifications' meaning, as redundant specifications do not constrain the program any more than the given non-redundant specifications. Concepts were left out if at most one tool supported them, taking into account tools whose support may be added in the future, such as KeY [1] and Verifast [25]. If several tools support a feature in some way, it is incorporated into the IR to facilitate translating between such tools without losing functionality. Currently, this mostly concerns memory access permissions for tools that are based on separation logic, such as VerCors and Verifast. The IR is defined in more detail with a grammar in the appendix[1].

[1] Due to publisher constraints, the appendix was moved online after peer review, to https://doi.org/10.4121/73361fbb-2633-4011-b615-cce19d8ac196.

Limitations The Specification Translator translates as much as possible, however we cannot guarantee that the file will verify with the target tool. One possible reason is that a concept used in the original specification is not directly supported by the target tool, and the user has to find a different way to express the property. The tool rewrites specifications into a related concept wherever possible. Another reason for an unsuccessful verification after translation is that the target tool requires more extensive annotations. Where possible, the tool generates annotations for built-in assumptions, such as objects being non-null in OpenJML, but it does not generate completely new specifications.

We do not provide soundness guarantees for the translation. To do so, one needs to formalise the semantics of all tools involved, which is an effort out of scope for this work. Instead, we provide carefully considered translations and show their usefulness in practice. In the unlikely event that a substantial part of the specifications is dropped or altered semantically, we expect the user or target verifier to catch this during re-verification (see Section 4).

The Specification Translator only uses syntactic analysis to identify features and translate them, there is no semantic analysis such as name resolution or typing. While this limits the translation in a few cases, as mentioned in Section 3, it significantly reduces the complexity of the tool, and thereby the potential for errors. The evaluation in Section 4 shows that this analysis is sufficient to translate the annotations in the overwhelming majority of cases.

Extending the tool The Specification Translator has been designed keeping extendability in mind. By having an IR, a new language does not require a direct translation to every other language. Instead, to add a new input language, one only needs to add a parser and a translation into the IR. To add a new output language, one needs to add a translation from the IR into the new output language. The syntax trees for the different tools are based on a common underlying data structure, on which any new tool-specific AST can also build. This eliminates the need to redefine AST nodes from scratch if similar nodes exist in the other languages.

Artifact The tool is available at https://doi.org/10.4121/21e79524-40c4-4dc1-8108-94e7b6fc6d9f.

3 Translating Annotations

The Specification Translator currently supports the translation of annotations between OpenJML, Krakatoa and VerCors. This section discusses for each tool the choices that have been made in the translations to and from these tools.

3.1 OpenJML

OpenJML [13] is an open-source tool, which can verify Java programs that are annotated with JML specifications. Its annotation language closely adheres to the JML standard. As the IR is largely based on JML, the translation from OpenJML to the IR does not require many changes.

Non-null by default One notable exception is that OpenJML assumes by default that references are not null, for instance a method only returns non-null values. References that can be null need to be explicitly annotated as `nullable`. This is different from other tools like VerCors and the Java standard, where references are always nullable. Therefore, the Specification Translator annotates all classes as `nullable_by_default` when translating to OpenJML, disabling this implicit assumption of OpenJML. In turn, when translating from OpenJML to the IR, the Specification Translator generates `non_null` modifiers in cases such as parameters that are nullable in Java but implicitly non-null in OpenJML.

Access permissions When translating to OpenJML, the Specification Translator needs to deal with the extensions that the IR adds to JML. One example are access permissions which are used by tools built on separation logic.

Access permissions are used to indicate whether it is allowed to read from or write to a variable. The Specification Translator will therefore generate `assignable` clauses for all variables that occur in permission expressions in preconditions, and `loop_modifies` clauses for all permission expressions in loop invariants. Tools based on fractional permissions [8], such as VerCors, allow for more fine-grained control than these JML clauses and therefore the translation can be an over-approximation.

If the permission expression also contains the value at that location, such as the "x points to 5" ($\texttt{x}\mapsto\texttt{5}$) of classical separation logic, then this is turned into a Boolean equality `x==5`. Other concepts of separation logic are also turned into Boolean versions, for instance the separating conjunction becomes a Boolean conjunction. This retains a significant part of the meaning, although additional annotations may be required before the program can be verified again, e.g. explicitly stating that two references are not aliases.

Privacy modifiers OpenJML takes privacy modifiers into account for its verification. For example, the contract of a public method usually cannot refer to private variables of the object. Other tools, like VerCors, do not have such a restriction, meaning everything is implicitly in the public scope. To mimic such behaviour, and avoid OpenJML giving many warnings about object visibility, the translation from the IR to OpenJML's AST marks all fields and methods, which are not already public, as `spec_public`.

Predicates Both Krakatoa and VerCors have *predicates*, with slightly different meanings. In Krakatoa, they are boolean functions (see Listing 1.1 for an example), while in VerCors they can also contain access permissions. Additionally, VerCors treats them like abstract functions and does not automatically inspect and use their body; an explicit `unfold` statement is needed to do so (for more details, see [32]). OpenJML does not support predicates like that. The most similar construct in OpenJML is a model method, which is a method that only exists for the specification. We can translate the declaration of the predicate into the declaration of a model method with a boolean return type and otherwise the same signature as the predicate. In Krakatoa and VerCors, predicates

Listing 1.1. Krakatoa predicate

```
/*@ predicate Sorted(int a[],
      integer l, integer h)
  @   = \forall integer i;
  @     l<=i<h ==> a[i] <= a[i+1];
  @*/
```

Listing 1.2. Translated predicate

```
/*@ ensures \result ==
  @   (\forall int i;  l <= i < h
  @     ==> a[i] <= a[i+1]);
  @ model boolean Sorted(int a[],
  @     int l, int h);
  @*/
```

do not have contracts and instead their body is completely visible to the calling context (potentially after using `unfold`). To mimic that, the body is turned into a postcondition of the model method. We show a small example of a predicate translation from Krakatoa to OpenJML in Listings 1.1 and 1.2.

Data types Some tools have additional data types that can be used in ghost code, such as VerCors' multisets. Other tools often do not have an exactly equivalent type, so the Specification Translator comments out any explicit reference to data types that OpenJML does not support. However, as our syntactic analysis does not do type checking, it cannot identify all places where an object of such a type is used. As a result, the Specification Translator may only comment out some of the usages, and others may require manual intervention.

3.2 Krakatoa

Krakatoa was originally developed as a Java frontend for the Why platform [17]. Nowadays, it can still be used in combination with Why3 [18]. It can verify Java programs annotated with the Krakatoa Modeling Language (KML) [34]. KML is inspired by JML and ANSI/ISO C Specification Language (ACSL). We have used Krakatoa's documentation and the generated WhyML programs to determine the semantics of KML. Krakatoa is no longer actively developed and there are only a handful of examples available which seems to indicate limited use of the tool. As a result, it seems that there is little demand for translation to Krakatoa, therefore we have chosen to provide limited support for this.

Assumes in behaviors Method contracts in JML can have *behaviors*. These clauses can be used to split the specification into multiple cases, e.g. if an element is in the list or not. To distinguish which case applies, Krakatoa uses *assumes*-clauses instead of *requires*-clauses. For the translation of assumes clauses into the IR, we use the semantics as explained in Krakatoa's reference manual[2]. Namely, given an assumes-clause A and an ensures-clause E in a behavior, then `\old` $(A) \implies E$ should hold if the program terminates normally.

Inductive predicates Krakatoa supports inductive predicates. These inductive predicates consist of a predicate, possibly some parameters, and several case definitions. The case definitions describe when the predicate should evaluate

[2] https://krakatoa.lri.fr/krakatoa.html

to true. This should be a least fixpoint, meaning that only these cases should evaluate to true and no others. Other tools do not have a concept to express a least fixpoint and therefore the best we can do is to over-approximate for the IR. We define a (non-inductive) predicate and translate the case definitions into postconditions. We warn the user about this over-approximation.

Lemmas Lemmas are typically used to assist the prover to determine whether the program adheres to the specification. They become part of the prover's implicit knowledge, and the prover can automatically use them where needed. We considered translating lemmas to ghost functions. However, that would require the user to call the ghost method explicitly, i.e. add additional annotations. Instead, we translate them into axioms in the IR which makes sure that the user does not need to add any annotations manually. Axioms are assumed to be correct, whereas lemmas generate proof obligations. The lemmas (should) have already been verified in Krakatoa originally, so simply assuming their correctness at this point should not introduce any unsoundness. Nevertheless, to be safe, we warn the user that these axioms should be proven separately.

3.3 VerCors

VerCors [6] is an open-source tool to verify concurrent programs, using JML-like annotations and permission-based separation logic as its foundation. While the specification language is JML-like, there are some notable deviations. In particular, many JML features are not supported, such as axioms, and instead there are new constructs to accommodate concepts from separation logic.

Permissions VerCors does not support `assignable` or `accessible` clauses in method contracts, which can be used to indicate whether you can write to or read from a variable. In VerCors this can be expressed with write permission for the locations that are assignable, and at least read permission for those that are accessible. Thus each `assignable` or `accessible` clause can be represented with a pair of a pre- and a postcondition containing those access permissions. Likewise, `loop_modifies` clauses are turned into loop invariants with write permissions. This may be an over-approximation as `loop_modifies` can refer to local variables on the stack, while permissions can only refer to heap locations. However, the Specification Translator's lack of name resolution means that we cannot distinguish them, and the user needs to remove these permissions manually. In the examples in the evaluation (Section 4), this was not a frequent case, and was always a quick and straight-forward fix.

Bound checks A notable difference between VerCors and other tools are type bounds: VerCors only supports unbounded integers. In contrast, OpenJML checks at every assignment if the value is within the bounds of a machine integer, and warns about potential over- or underflows. This means that a program that verifies in VerCors may not verify in OpenJML. In the opposite direction, a

verified OpenJML program will also verify in VerCors, as long as the specification does not explicitly rely on the boundedness guarantees. The Specification Translator could try to add explicit assertions about value bounds to every assignment to mimic OpenJML's behaviour. However, they would require information about the type of expressions to derive the right bounds, which the syntactic analysis of the Specification Translator cannot always provide. More importantly, they would clutter the program a lot, so we decided to not include those checks. Instead, the user has to manually add bound checks when translating to VerCors, if variable bounds are of interest. Moreover, they may have to provide additional annotations that ensure variable bounds after translating from VerCors to OpenJML, before the program verifies again, or disable the bound checks in OpenJML.

Behaviors Unlike the IR, VerCors does not support `behavior`-clauses. In other tools, like OpenJML and Krakatoa, a method contract can specify multiple behaviours for the method, for example depending on the value of a parameter (Listing 1.3 gives an example). Whenever a `behavior`'s precondition is met, the method has to adhere to the specifications of that `behavior` block, such as guaranteeing its postconditions. When calling the method, at least one `behavior`'s precondition has to be met. In contrast, in VerCors all preconditions must be met at every call, and the postconditions are ensured unconditionally. However, a postcondition can be an implication, thereby explicitly containing a condition.

The Specification Translator uses this to turn a `behavior` into conditional postconditions: Each postcondition of a `behavior` is turned into an implication, using the conjunction of all preconditions of the `behavior` as a condition (see Lines 8-9 and 10-12 in Listing 1.4). If the `behavior` contains `accessible` and `assignable` clauses, they are treated similarly: They are turned into a pair of pre- and postcondition as described above, both of which are conditional on the `behavior`'s precondition (Lines 1-4). Additionally, the individual preconditions of the `behavior` are replaced with one single precondition of the form $\bigvee_{\texttt{behavior } b} \left(\bigwedge_{\text{precond. } p \text{ in } b} p \right)$, meaning a disjunction over all `behavior` blocks for the method, where each disjunct is a conjunction of the preconditions of that `behavior` (Lines 5-7). If the clause used the keyword `normal_behavior` or `exceptional_behavior`, then this clause implicitly assumes the postcondition `signals (Exception e) false` or `ensures false`, respectively. These implicit postconditions are made explicit in the translation, and also become conditional on the `behavior`'s preconditions (Line 13).

Invariants In JML, one can define invariants, which an object has to satisfy at every observable execution point, such as the end of initialisation and before and after invoking any method which is not declared as `helper` [28, Ch. 8.2]. VerCors does not support invariants. Thus, invariants are turned into pre- and postconditions for every non-`helper` method, and postconditions for constructors. While this is a close approximation, it does not exactly replicate the meaning of `invariant`: Observable points also include any time when no method is ongoing. The authors of [28] note themselves that the definition is highly non-modular,

Listing 1.3. OpenJML program with `behaviors`

```
1  /*@ behavior
2    requires inp > 0 || inp==0;
3    requires inp > -1;
4    ensures \result;
5  also exceptional_behavior
6    requires inp < 0;
7    assignable errors;
8    signals(Exception e)
9      errors == \old(errors)+1;
10 @*/
11 boolean checkPos(int inp){
12   ...
13 }
14
```

Listing 1.4. Translated contract

```
1  /*@ requires(inp<0)
2        ==> Perm(errors,write);
3      ensures \old(inp<0)
4        ==> Perm(errors,write);
5      requires ((inp>0 || inp==0)
6                && (inp>-1))
7               || (inp<0);
8      ensures \old((inp>0 || inp==0)
9               && (inp>-1)) ==> \result;
10     signals(Exception e)
11       \old(inp<0) ==>
12         (errors == \old(errors)+1);
13     ensures \old(inp<0) ==> false;
14 */
15     boolean checkPos(int inp) {...}
```

and propose the same work-around we use, to allow modular verification. Note that in the concurrent setting of VerCors, a concurrent thread may observe more execution points than the ones mentioned above. Zaharieva-Stojanovski et al. [35] propose a more involved notion of class invariants for concurrency. However, this requires explicitly marking program segments where the class invariant may be broken. VerCors does not support these special annotations, yet, so we keep the original notion of observability.

Predicates As mentioned in Section 3.1, VerCors supports predicates, which can contain both boolean expressions and access permissions. To use this content, the predicate has to be explicitly unfolded, and refolded back into a predicate afterwards. In the translation from VerCors to the IR, these `fold` and `unfold` statements are turned into assertions. This mimics the fact that VerCors actually checks that the predicate holds at that position. It can also serve as a guidance to provers, indicating that the knowledge of this predicate is needed here.

4 Evaluation

We evaluate how much the Specification Translator improves tool interoperability by showing (1) how many annotations can be reused between tools, and (2) how the Specification Translator allows the reuse of tools like Daikon.

4.1 Reuse of Specifications

In this section we focus on the question "How many annotations can be reused?". We have randomly selected 10 verifiable programs per tool (Krakatoa, OpenJML and VerCors) from the Java examples that are distributed with each tool, as well as 2 bigger case studies. We will first discuss the results for the smaller examples. This includes overviews per tool of how many annotations could be translated, a discussion on how much manual effort is needed after translation and a note on how many translations are trivial.

For each program, we measure how many lines were translated. This is determined by inspecting the specifications to see whether the intent of the specification before translation was preserved in the specification after translation. This could either be through a correct translation, or by omitting the specification if there is a corresponding default assumption in the target verifier. Aside from how many lines could be translated, we also measure how long the translation took, and whether the program could be verified after translation. If the verification was unsuccessful, the error message was manually inspected to determine the cause of the verification failure.

We distinguish between errors caused by the verifier (e.g. missing support for a Java construct used in the original file) and those caused by the translator (e.g. a method signature is no longer uniquely defined after translation). For the verification, we have used OpenJML v0.8.59 and VerCors v2.0.0 (beta). As mentioned in Section 3.2, support for translating to Krakatoa is limited, so we did not evaluate that direction. However, all Krakatoa examples were verified with Krakatoa v2.41 (with Why3 v0.88.3) before translating them. For the translation time, each translation was run five times, and the average is provided. The given time is CPU time, obtained with Python profiling tools. The time was recorded in the virtual machine which was provided for the iFM 2023 artifact evaluation (4 CPU cores, 8 GB of RAM, running Ubuntu 22.04).

The results can be seen in Tables 1 to 3. The second column of the tables indicates the number of lines that have specifications. This includes lines of code that have specifications embedded, such as `void m(/*@nullable*/ int[] a)`, but does not include lines that have no meaningful specifications, such as the line `@*/`. Of the total of 991 lines analysed for this subsection, 806 were successfully translated or not needed in the target tool (81%).

Krakatoa examples The results of translating the Krakatoa examples to OpenJML and VerCors can be found in Table 1. The translation takes around three seconds for each example, which is significantly faster than any human could translate them. Several examples verify after translation without any manual intervention.

There are several examples (tagged 'I') that require additional annotations before they can be re-verified. For example, OpenJML checks for integer overflows while this was disabled in several Krakatoa examples. Therefore, OpenJML requires additional specifications about the bounds of variables. The `MyCosine` example only had annotations that used built-in functions from Krakatoa. These functions are not available in other tools and can therefore not be translated. Nonetheless, the translation to OpenJML contains some annotations as the tool adds privacy modifiers and ensures that objects are nullable as discussed in Section 3.1. The translation introduces an error in one case (`TreeMax`), which is a name clash between two originally polymorphic methods. This is a limitation of the Specification Translator as it does not do name resolution.

Most of the issues we ran into when verifying programs with VerCors after the translation were caused by limitations of VerCors (tagged 'L'). Some were caused by minor bugs in VerCors, such as missed corner cases of the parser,

Table 1. Krakatoa examples and their translation into OpenJML and VerCors. In the result columns, 'I' indicates incomplete, i.e. more annotations are needed. 'E' indicates empty, i.e. the file does not contain any annotations that express the intent of the original annotations. 'T' indicates that the translation introduced an error. 'L' indicates failed verification due to a limitation of the target verifier.

Krakatoa		OpenJML				VerCors			
program	lines with annotations	lines successfully translated	lines with annotations	time in s	result	lines successfully translated	lines with annotations	time in s	result
`ArrayMax`	13	100%	12	2.67	✗(I)	85%	9	2.79	✗(I)
`BankingExample`	6	100%	8	2.12	✓	83%	7	2.16	✗(L)
`Counter`	5	40%	5	3.00	✗(I)	40%	2	2.19	✗(I)
`Creation`	20	95%	21	3.11	✗(I)	75%	7	2.16	✗(L)
`MyCosine`	9	0%	5	2.61	✓(E)	0%	0	2.19	✗(L)
`Negate`	9	89%	10	2.98	✗(I)	78%	8	2.23	✗(L)
`Purse`	16	94%	19	2.66	✗(I)	88%	18	2.22	✗(I)
`Sort2`	38	11%	7	2.93	✗(I)	11%	4	2.63	✗(L)
`Termination`	6	83%	9	2.66	✓	83%	5	2.12	✓
`TreeMax`	24	33%	15	2.65	✗(T)	25%	5	2.40	✗(L)

that we expect to be fixed in the near future and thus we have not modified our tool. VerCors also has limited support for inheritance in Java and difficulties dealing with `Class.method()` calls. Similar to the previous examples, the user sometimes needs to add additional annotations (tagged 'I').

OpenJML examples Table 2 shows that all OpenJML examples could be translated to VerCors, with only a few lines not being translated. One such line had modifiers to disable integer bound checking that VerCors does not perform anyway, and one only contained a `nullable` modifier that is already the default in VerCors. Thus, the only line with an effect is a loop invariant that referred to the OpenJML built-in variable `\count` which does not exist in VerCors.

Like before, the translation time was negligible. Unfortunately, none of the examples verified after translation. This is to be expected, as VerCors requires the additional information of access permissions. While the Specification Translator tries to derive them from e.g. `assignable` clauses, none of the examples used those clauses. All the examples marked with 'I' were missing access permissions, and verified after manually adding those. Several examples failed because of limitations of VerCors, e.g. it does not support calls to standard library functions. Examples failing because of such limitations of VerCors are marked with 'L'. None of the verification failures are caused by the translator, instead they are caused by limitations of the verifiers.

Table 2. OpenJML examples and their translation into VerCors. In the result columns, 'I' indicates incomplete, i.e. it needs more annotations to be verified. 'L' indicates that the program could not be verified due to a limitation of the verifier.

OpenJML		VerCors			
program	lines with annotations	lines successfully translated	lines with annotations	time in s	result
`BinarySearchGood`	12	100%	12	2.70	✗(L)
`BubbleSort`	10	100%	11	2.72	✗(I)
`ChangeCase`	8	100%	4	2.53	✗(L)
`HeapSort`	46	100%	47	3.80	✗(L)
`InvertInjection`	16	100%	16	3.49	✗(L)
`MaxByElimination`	7	100%	8	3.49	✗(I)
`MergeSort`	19	100%	21	3.01	✗(L)
`SelectionSort`	12	100%	12	2.75	✗(I)
`SumMax`	4	75%	4	2.73	✗(L)
`TwoSum`	18	84%	18	3.05	✗(L)

VerCors examples The results of translating the VerCors is shown in Table 3. Some VerCors examples verified successfully in OpenJML. One example used a public class which did not match the file name, so OpenJML failed to verify it. This highlights the stronger visibility checks of OpenJML. Renaming the file made it verify. Several examples caused OpenJML to issue warnings about potential over- or underflows, but did otherwise verify. In some cases, constructs were not supported, such as data types, and additional annotations are required.

Effort to Verify We also investigated how much manual effort is needed after the automatic translation, before the files verify successfully in the respective target tool. In some cases, the effort was negligible, for instance all files in Table 3 marked *overflow* verify if the overflow check is turned off by adding the modifiers `code_bigint_math` and `spec_bigint_math` to the respective class. Notice that turning off these checks actually makes the behaviour of OpenJML more faithful to the behaviour of VerCors. However, the translator keeps the checks by default, as this allows the user to leverage the additional capabilities of the target tool.

In other cases, only a little more effort is required. For Krakatoa, six of the translations to OpenJML required three or less changes, e.g. adding a bound check or redefining a simple predicate. Six translations from Krakatoa to VerCors also required some effort. For translations to VerCors, most examples required permissions to be added manually. For those small example files, an experienced VerCors user can determine the necessary permissions easily. However, in general this can be a non-trivial task, and research into permission inference is still ongoing (e.g. [14]). In such cases, one could use the Specification Translator in a workflow pipeline, followed by invoking a dedicated inference tool. Another common task for VerCors was to add an existing precondition also as a postcon-

Table 3. VerCors examples and their translation into OpenJML.

VerCors		OpenJML			
program	lines with annotations	lines successfully translated	lines with annotations	time in s	result
`BoogieTest`	6	100%	7	2.39	✗(overflow)
`Incr`	6	100%	11	2.41	✗(overflow)
`KnuthTabulate`	8	100%	15	2.49	✗(incomplete)
`LabeledWhile`	19	63%	23	2.41	✓
`ListAppend-ASyncDefInline`	17	38%	9	2.53	✗(incomplete)
`LoopInv`	4	100%	5	2.34	✗(overflow)
`PairInsertionSort`	32	100%	42	2.87	✓
`refute4`	3	100%	6	2.25	(✓) (name)
`RosterFixed`	24	88%	32	2.61	✗(limitation)
`SwapInteger`	6	100%	11	2.25	✓

dition or loop invariant. While the need for this may not always be obvious, an experienced user will quickly recognise and resolve the issue.

There were also five examples that required significant user intervention, such as redefining an intricate predicate. For the Krakatoa examples, these were `Sort2` and `TreeMax`, which required significant effort to rewrite predicates. For the VerCors examples, in `ListAppendASyncDefInline` a VerCors-specific data type needed to be replaced, and `RosterFixed` required defining a JMLDataGroup to handle access restrictions. To resolve the issues for these VerCors examples, we also used a newer version of OpenJML (0.17.0-alpha-15). For the OpenJML examples, only `MergeSort` needed complex user intervention, namely providing a contract for a Java library function.

Some files could not be verified at all, because they relied on features not supported by the target tool. For example, VerCors does not support some Java features used by two Krakatoa examples and two OpenJML examples.

Trivial Translations As many specification languages are based on JML, we also analysed how many of the translations were trivial, i.e. required no change at all for reuse. We compared the specifications generated by the Specification Translator with the original ones. We found that around 55% of the original specification lines needed changing in order to be reused, or were commented out because there was no direct translation. Thus, if translated manually without our tool, these 55% would require some user intervention. Since the Specification Translator was able to translate 81% of specification lines, we conclude that using it significantly reduces the amount of user intervention required.

Case Studies Next, we demonstrate the Specification Translator on two bigger API verification case studies.

Firstly, we use a verification case study that used Krakatoa to verify a genetic algorithm [9]. The original file has 164 lines with specification. We have translated these specifications to OpenJML, resulting in 178 lines with specification. We were able to translate 89% of the original specification. Before OpenJML can analyse the file, some additional annotations are needed to resolve parsing issues. The original file contained several predicates that could not be translated, as they used labels, therefore the user will have to redefine these. The user will also need to add some `static` modifiers to several of these predicates. Finally, several `\fresh` expressions are used in preconditions. These should be removed as OpenJML does not allow `\fresh` expressions in preconditions.

Secondly, we use a verification case study where a version of Java's ArrayList was verified in VerCors[3]. The VerCors file has 258 lines with specifications. We used the Specification Translator to translate this to OpenJML, with the resulting file having 327 lines with specifications. All the lines could be translated. OpenJML could parse the translated file without manual intervention, and already verified 17 out of 23 methods successfully. Some of the warnings resulted from two pure methods not being marked `pure`. Marking these methods as `pure` reduced the number of OpenJML warnings to 10. The majority concerned over- or underflows in arithmetic operations, which is to be expected when translating from VerCors. Another is related to inheritance, which is also not fully supported by VerCors. One method was a stub stand-in for a library, and used as a body `assume false`. OpenJML warned about that, but had no other complaints about that method. Finally, the anti-aliasing of VerCors' separation logic did not fully translate to OpenJML, causing some warnings.

To conclude, for the Krakatoa and VerCors case studies we were able to translate 89% and 100% respectively. This shows that using the Specification Translator allows us to reuse large parts of existing specifications for APIs, and significantly reduces the manual effort required for translation.

4.2 Reuse of Tools

Aside from reusing specifications, the Specification Translator also allows reusing tools that only support a limited number of specification languages. For example, Daikon [16] can generate specifications in JML, but does not support VerCors' specification language. With our tool, the JML specifications generated by Daikon can be translated to VerCors. This makes Daikon applicable for verifiers that do not support JML directly, without having to alter Daikon.

As an example, we use the `QueueAr.java` provided with Daikon. Running Daikon on the un-annotated file generates 127 lines with specifications. Many of those are JML-specific and not directly usable in VerCors, in particular class-level invariants and `assignable` clauses. After using the Specification Translator, the file had 252 lines with annotations. 16 of those referred to the `owner` of JML's type system, which VerCors does not support, but the remaining 94% were valid VerCors specifications. Note that this means an increase by more than

[3] This was done by student Joost Sessink as part of a course.

100 usable specification lines. These mainly stem from adding class-level invariants as pre- and postconditions to all relevant methods. Without the Specification Translator, all these specifications would have to be written manually.

Note that while the specifications generated by Daikon were helpful, they were not sufficient for verification with OpenJML, nor with VerCors after the translation. However, our goal was to make Daikon-generated specifications usable for VerCors, regardless of the verification result. This was successful for nearly all specifications.

5 Related Work

To the best of our knowledge, this paper presents the first tool for translating annotations for deductive verification. We discuss three related research areas: (1) translations to/from JML, (2) common tool formats, and (3) tool interoperability.

Translating to/from JML Many translations to and from JML have been proposed in earlier work. This includes translation from B machines to JML [10], OCL to/from JML [23], JML to executable Java [4], JML from Alloy expressions [21], from temporal properties [19], from VDM-SL [33] and from security automata [24]. However, none of these target the translation of annotations between different deductive verifiers. Also of note is Raghavan and Leavens [29], who simplify JML specifications by removing syntactic sugar, thus a JML-to-JML translation. While some of the transformations we do are similar, there is not a lot of overlap: Many of the syntactic sugar that they remove is already supported by the verifiers we looked at. Moreover, their translation left concepts that we needed to remove, such as `behavior` clauses when translating to VerCors.

Common tool formats Many different common formats for tools have been proposed, such as the earlier mentioned JML (Java Modeling Language) [27]. Other formats include JIR (JML Intermediate Language) and JFSL (JForge Specification Language). JIR [31] aims to decouple front-ends and back-ends by introducing an intermediate representation. JFSL aims to address some shortcomings of JML by extending it with e.g. support for expressing the transitive closure [11]. These languages are not sufficient for our goals since they do not support constructs outside of JML such as permissions in VerCors. Moreover, we believe that a new language will not solve the problem because new techniques are still being developed that may require new types of specifications. Also, it would require all tool developers to modify their tool to support the new language. With the Specification Translator we can improve the tool interoperability between deductive verifiers without burdening the tool developers.

Tool interoperability Another related topic is tool interoperability. Christakis et al. [12] have proposed to extend the output of static verifiers to make implicit

assumptions explicit, for instance about integer overflows. They then use other verifiers or test case generation to check these assumptions, creating a tool chain. During our translation, we similarly turn some implicit assumptions into explicit proof obligations. While we both address tool interoperability, they focus on the verification results and how these can be used in tool chains, whereas we focus on the reuse of specifications between verifiers as well as from inference tools such as Daikon. Aside from making assumptions explicit, our approach also supports the reuse of other specifications through translation. Consequently, our target verifier attempts to re-verify the properties proven by the original verifier, without reusing its results. Moreover, our approach does not require any modifications to the existing verifiers as is the case for the work of Christakis et al. For their approach, differences in the specification languages of tools can be a major hindrance, and a translation tool like ours can increase the applicability of their work.

Another idea that focuses on the reuse of existing artefacts is proof repair. The idea of proof repair is to automatically update proofs used by proof assistants. This can be used to fix a proof when a newer version of the same proof assistant has changes that are backward incompatible [30]. Instead of updating proofs, we focus on updating specifications between different deductive verifiers. We can extend the Specification Translator to support translating specifications between different versions of the same tool to achieve a similar goal for deductive verifiers.

6 Conclusion

We have presented the Specification Translator, the first tool for translating contract-based specifications between deductive verifiers for Java. Using the Specification Translator allows users to reuse existing tools and specifications, thereby minimising the burden of writing specifications and enhancing tool interoperability. We have shown that we could translate 81% of the specifications between tools, reducing the effort for the user significantly compared to having to translate them manually. Moreover, it allows us to use tools such as Daikon for new deductive verifiers without needing to modify these tools.

For future work, aside from supporting other tools like Verifast, we would like to exploit the new abilities for reuse. This includes building verified libraries based on verification efforts from other tools, and exploring tool integration like the Daikon example.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification – The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer International Publishing (2016). https://doi.org/10.1007/978-3-319-49812-6, Tool website: https://www.key-project.org/

2. Armborst, L., Huisman, M.: Permission-based verification of red-black trees and their merging. In: 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE). pp. 111–123 (2021). https://doi.org/10.1109/FormaliSE52586.2021.00017
3. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification — specification is the new bottleneck. Electronic Proceedings in Theoretical Computer Science **102**, 18–32 (nov 2012). https://doi.org/10.4204/eptcs.102.4
4. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12476, pp. 60–80. Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_4
5. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) Verified Software. Theories, Tools, and Experiments. pp. 35–48. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
6. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S.A. (eds.) Integrated Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10510, pp. 102–110. Springer (2017). https://doi.org/10.1007/978-3-319-66845-1_7, Tool website: https://www.utwente.nl/vercors/
7. Boer, M.d., Gouw, S.d., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal specification and verification of JDK's identity hash map implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods. pp. 45–62. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-07727-2_4
8. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) Static Analysis. pp. 55–72. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
9. Brizhinev, D., Goré, R.: A case study in formal verification of a Java program. Computing Research Repository **abs/1809.03162** (2018), http://arxiv.org/abs/1809.03162
10. Cataño, N., Wahls, T., Rueda, C., Rivera, V., Yu, D.: Translating B machines to JML specifications. In: Ossowski, S., Lecca, P. (eds.) Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012. pp. 1271–1277. ACM (2012). https://doi.org/10.1145/2245276.2231978
11. Chicote, M., Ciolek, D., Galeotti, J.: Practical JFSL verification using TACO. Software: Practice and Experience **44**(3), 317–334 (2014). https://doi.org/10.1002/spe.2237, https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2237
12. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012: Formal Methods. Lecture Notes in Computer Science, vol. 7436, pp. 132–146. Springer Berlin Heidelberg (2012). https://doi.org/10.1007/978-3-642-32759-9_13
13. 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). https://doi.org/10.1007/978-3-642-20398-5_35, Tool website: https://www.openjml.org/

14. Dohrau, J.: Automatic Inference of Permission Specifications. Ph.D. thesis, ETH Zurich (2022)
15. Efremov, D., Mandrykin, M., Khoroshilov, A.: Deductive verification of unmodified linux kernel library functions. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification. pp. 216–234. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_15
16. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The Daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015, Tool website: https://plse.cs.washington.edu/daikon/
17. Filliâtre, J., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 173–177. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_21, Tool website: https://krakatoa.lri.fr/
18. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems. ESOP. pp. 125–128. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
19. Giorgetti, A., Groslambert, J.: JAG: JML annotation generation for verifying temporal properties. In: Baresi, L., Heckel, R. (eds.) Fundamental Approaches to Software Engineering, 9th International Conference, FASE 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3922, pp. 373–376. Springer (2006). https://doi.org/10.1007/11693017_27
20. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 273–289. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
21. Grunwald, D., Gladisch, C., Liu, T., Taghdiri, M., Tyszberowicz, S.S.: Generating JML specifications from alloy expressions. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8855, pp. 99–115. Springer (2014). https://doi.org/10.1007/978-3-319-13338-6_9
22. Hähnle, R., Huisman, M.: Deductive software verification: From pen-and-paper proofs to industrial tools. In: Steffen, B., Woeginger, G.J. (eds.) Computing and Software Science - State of the Art and Perspectives, Lecture Notes in Computer Science, vol. 10000, pp. 345–373. Springer (2019). https://doi.org/10.1007/978-3-319-91908-9_18
23. Hamie, A.: Translating the object constraint language into the java modelling language. In: Proceedings of the 2004 ACM Symposium on Applied Computing. p. 1531–1535. SAC '04, Association for Computing Machinery, New York, NY, USA (2004). https://doi.org/10.1145/967900.968206

24. Huisman, M., Tamalet, A.: A formal connection between security automata and JML annotations. In: Chechik, M., Wirsing, M. (eds.) Fundamental Approaches to Software Engineering, 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5503, pp. 340–354. Springer (2009). https://doi.org/10.1007/978-3-642-00593-0_23
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: NASA Formal Methods Symposium. pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4
26. Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. Electronic Proceedings in Theoretical Computer Science **284**, 53–70 (nov 2018). https://doi.org/10.4204/eptcs.284.5
27. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for java. SIGSOFT Softw. Eng. Notes **31**(3), 1–38 (may 2006). https://doi.org/10.1145/1127878.1127884
28. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: JML Reference Manual (May 2013), dept. of Computer Science, Iowa State University. Available from http://www.jmlspecs.org
29. Raghavan, A., Leavens, G.: Desugaring JML method specifications. Computer Science Technical Reports **345** (2005), http://lib.dr.iastate.edu/cs_techreports/345
30. Ringer, T., Yazdani, N., Leo, J., Grossman, D.: Adapting proof automation to adapt proofs. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. p. 115–129. CPP 2018, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3167094
31. Robby, Chalin, P.: Preliminary design of a unified JML representation and software infrastructure. In: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs. FTfJP '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1557898.1557903
32. Summers, A.J., Drossopoulou, S.: A formal semantics for isorecursive and equirecursive state abstractions. In: Castagna, G. (ed.) ECOOP 2013 – Object-Oriented Programming. pp. 129–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39038-8_6
33. Tran-Jørgensen, P.W.V., Larsen, P.G., Leavens, G.T.: Automated translation of VDM to JML-annotated Java. Int. J. Softw. Tools Technol. Transf. **20**(2), 211–235 (2018). https://doi.org/10.1007/s10009-017-0448-3
34. Tushkanova, E., Giorgetti, A., Marché, C., Kouchnarenko, O.: Modular Specification of Java Programs. Research Report RR-7097, INRIA (2009), https://hal.inria.fr/inria-00434452
35. Zaharieva-Stojanovski, M., Huisman, M.: Verifying class invariants in concurrent programs. In: Gnesi, S., Rensink, A. (eds.) Fundamental Approaches to Software Engineering. pp. 230–245. Springer Berlin Heidelberg (2014). https://doi.org/10.1007/978-3-642-54804-8_16