# Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers

Sophie Lathouwers
University of Twente
Enschede, the Netherlands
s.a.m.lathouwers@utwente.nl

Marieke Huisman
University of Twente
Enschede, the Netherlands
m.huisman@utwente.nl

## ABSTRACT

Deductive verification can be used to ensure properties about all possible behaviours of a program, even when the program is parameterised and has an unbounded state space. But to achieve this, the user needs to specify what the desired properties are, and often needs to guide the prover with auxiliary annotations. This paper investigates what annotations are actually needed, and it provides a taxonomy to categorise these annotations. In particular, we identify several top-level categories, which are further divided into subcategories of annotations. This taxonomy is then used as a basis to investigate how often particular annotation categories occur, by inspecting over 10k lines of annotated programs. To determine whether the results are in line with expectations, we have interviewed several experts on deductive verification. Moreover, we show how the results can be used to evaluate the effectiveness of annotation generators. The knowledge from this analysis provides a gateway to guide further research in improving the efficiency of deductive verification, e.g.: it can serve as a guideline on what categories of annotations should be generated automatically, to evaluate the power of existing annotation generation techniques, and to improve the teaching of deductive verification.

## CCS CONCEPTS

• **Theory of computation** → **Program specifications**; *Automated reasoning*; *Pre- and post-conditions*; Invariants; • **General and reference** → *Empirical studies*; • **Software and its engineering** → **Specification languages**; *Formal software verification.*

## KEYWORDS

Taxonomy, Auto-active verification, Specifications, Annotations, Deductive verification

**ACM Reference Format:**
Sophie Lathouwers and Marieke Huisman. 2022. Formal Specifications Investigated: A Classification and Analysis of Annotations for Deductive Verifiers. In *International Conference on Formal Methods in Software Engineering (FormaliSE'22), May 18–22, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524482.3527652

## 1 INTRODUCTION

Writing correct software is notoriously difficult. Therefore, many techniques have been developed to help with this, such as fuzzing, model checking and static analysis. One of these techniques, deductive verification, is especially powerful to ensure software reliability because it works for all possible executions, for unbounded parameters, and for an unbounded state space. Deductive verification uses logical inference to determine whether a program adheres to a formally defined specification [17]. It generates proof obligations which need to be discharged either interactively (e.g. with a proof assistant like Isabelle) or automatically (e.g. with an SMT solver like Z3). At the end of the verification process, the deductive verifier will either (1) report that the program adheres to the specification, (2) report that it could not be proven correct, possibly showing what could not be proven, or (3) the verifier can timeout.

Unfortunately, while this technique is very powerful, it needs many specifications which can be difficult and time-consuming to write. As a result, writing specifications acts as a bottleneck in the deductive verification process [3, 4, 17]. An example of a program with specifications can be seen in Listing 1. The specifications are written in the form of *annotations* in the program code. The 10 lines of code in Listing 1 already require 7 lines of annotations to prove correctness.

**Listing 1: A small program that searches for an element in an array. Annotations (preceded by @) have been added so it can be statically verified with OpenJML.**

```
public class SearchArray {
  /*@ requires a!=null;
    @ requires a.length>0;
    @ ensures \result>=0 ==> a[\result]==elem;
    @ ensures \result==-1 ==> (\forall int i;
        0<=i && i<a.length; a[i]!=elem); */
  public static int search(int[] a, int elem) {
    int i = 0;
    /*@ loop_invariant 0<=i;
      @ loop_invariant i<=a.length;
      @ loop_invariant (\forall int j; 0<=j &&
          j<i; a[j]!=elem); */
    while (i<a.length) {
      if (a[i]==elem) { return i; }
      i++;
    }
    return -1;
  }
}
```

To improve the deductive verification process, we need to alleviate the specification writing burden on the user. Many researchers therefore suggest to look into the automatic generation of annotations [13, 16, 18, 20, 27]. But before we can effectively generate annotations, we need to improve our understanding of what annotations we should generate. Specifically, we need to know:

- What annotations are used?
- What is the role of these annotations? (e.g. do they describe the behaviour of the program or are they used to help the underlying solvers?)
- How frequently do these annotations occur?

This information can be used in many ways to improve the deductive verification process, for example:

- Guide research into annotation generation, e.g. by focusing on the most common annotations.
- Evaluate existing annotation generation techniques.
- Guide teaching deductive verification based on the common annotations amongst tools.
- Evaluate the current state of the art for deductive verification, e.g. it can be used to investigate what common features of a language are lacking support in the verifiers.

While there has been some related work that analyses specifications [10, 12, 28], these focus mostly on lightweight specification approaches such as debugging. We present the first analysis of all annotations that users need to write to statically verify a program.

We focus on deductive verification with automatic discharging of proof obligations, also known as auto-active verification [23]. In auto-active verification the user is only required to write specifications in the form of annotations in the program code, the rest of the verification process is automatic. We have chosen to focus on auto-active verification because the annotation writing process seems easier to (partly) automate than the user interaction required by interactive theorem provers.

This paper investigates which annotations exist and how often these are used by auto-active verifiers. We have set up a taxonomy which categorises annotations and provides an overview of the different types of annotations that are used. This taxonomy is based on input from verification experts and literature. Moreover, to gain insight into what auto-active verifiers need for proving correctness of programs, we categorised the annotations of a large data set of verified programs using the taxonomy. Afterwards, we have conducted interviews with verification experts to determine whether our results are in line with their beliefs. Additionally, we show how the results can be used to evaluate the impact of existing annotation generators.

To make this work feasible, the scope of this research has been limited to verifiers for Java programs that use preconditions, postconditions and loop invariants to write annotations. Based on these criteria, we included examples from the following verifiers: KeY [1], Krakatoa [24], OpenJML [8], VerCors [5] and Verifast [19].

In summary, the contributions of this research are:

- A taxonomy for categorising annotations of auto-active verifiers for Java programs.
- An analysis, based on the taxonomy, of annotations used by five auto-active verifiers (KeY, Krakatoa, OpenJML, VerCors and Verifast).
- A data set of annotations from verified Java programs categorised according to the taxonomy: https://doi.org/10.4121/16545714
- An evaluation of the impact of seven existing annotation generators on the overall verification process, based on the taxonomy and analysis.

## 2 DETERMINING CATEGORIES FOR THE TAXONOMY

To set up a taxonomy, we need to look for possible categories into which the data can be divided. This section describes our method for finding the categories for the taxonomy described in Section 3.

We have used open card sorting to determine initial categories for this taxonomy. In open card sorting, a participant is given multiple cards and is asked to sort these cards into groups that make sense to him/her and label each category [30]. This gives insight into what users think are similar cards. By using open card sorting, we consider the views of other researchers and avoid a personal bias in chosen categories. Moreover, card sorting requires little time per participant and is easy and inexpensive to set up.

The process of making the cards is explained in Section 2.1. Section 2.2 explains how we have set up our online card sort experiment, and presents the results of the card sort.

### 2.1 Making cards

We wanted to use existing annotations for the cards. Therefore, we started by choosing verifiers from which we could gather annotation samples. We have used the following selection criteria. The tool:

- can verify Java programs
- supports auto-active verification
- can still be used successfully (so we can reproduce the verification results)
- uses pre-/postcondition structure to write annotations

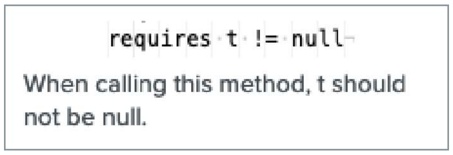
The final set of tools that are used for this study are: *KeY*[1], *Krakatoa*[2], *OpenJML*[3], *VerCors*[4], and, *Verifast*[5]. KeY and Krakatoa can be used with and without significant user interaction. For this research we only considered programs that can be verified without any significant user interaction (i.e. user only needs to press a button).

Next, we extracted the annotations from 10 randomly selected verified examples per tool. It is recommended to have at most 30-50 cards for an open card sort to avoid participant fatigue. Therefore, from the randomly selected examples, we selected annotations with the most commonly used constructs such as preconditions, postconditions, loop invariants and predicates. For each of these constructs, we made sure to add several cards to ensure that participants could form groups. This resulted in 30 cards.

In our card sorting, each card consists of one annotation and a corresponding description of the annotation to ensure a common understanding as recommended by [32] (see Figure 1 for an example

[1]v2.6.3 with CVC3 (v2.4.1) as a backend solver.
[2]v2.41 with Alt-Ergo (v1.30), CVC3 (v2.4.1) and Z3 (v4.3.2) as backend solvers.
[3]v0.8.46-20200505 with Z3 (v3.4.3) as a backend solver.
[4]commit 0aa6a9a, https://github.com/utwente-fmt/vercors/tree/0aa6a9a
[5]v19.12.06-17-g996506d

of a card). For the descriptions we were careful to vary wording to avoid obvious patterns for the participants.



```
requires t != null
```

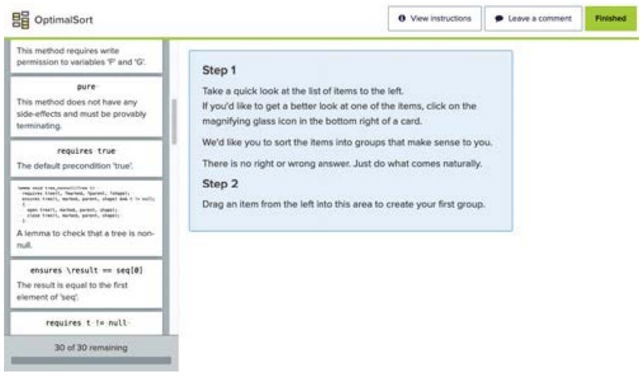
When calling this method, t should not be null.

**Figure 1: Example of a card used for card sorting in this study.**

## 2.2 Card sorting

Six people have participated in our open card sorting. The participants had varying degrees of experience with auto-active verification, some have verified small examples whereas others have done large case studies with such tools. The verifiers that they had experience with included Frama-C, KeY, Nagini, OpenJML, VerCors and Verifast. In our case it is not necessary to have a large amount of participants because it is used as a brainstorming exercise, not to gather statistically significant data.

Participants were given all 30 cards that they needed to sort into categories using a drag-and-drop approach (see Figure 2). The cards were presented in a random order to avoid any bias that might be introduced by the participants seeing the cards in the same order.



This method requires write permission to variables 'F' and 'G'.

```
pure
```

This method does not have any side-effects and must be provably terminating.

```
requires true
```

The default precondition 'true'.

[illegible]

A lemma to check that a tree is non-null.

```
ensures \result == seq[0]
```

The result is equal to the first element of 'seq'.

```
requires t != null
```

30 of 30 remaining

Step 1

Take a quick look at the list of items to the left.
If you'd like to get a better look at one of the items, click on the magnifying glass icon in the bottom right of a card.

We'd like you to sort the items into groups that make sense to you.

There is no right or wrong answer. Just do what comes naturally.

Step 2

Drag an item from the left into this area to create your first group.

**Figure 2: Participant's view during card sorting. The participant can drag and drop the cards from the left sidebar into the bigger area on the right to sort them into groups.**

### 2.2.1 *Card sorting results.*

Next, we analysed the results of the card sort. If participants had a group with a similar title and similar cards, then these groups were merged, since such groups indicate that these users have a similar understanding of these annotations. Participants suggested the following categories:

(1) Functional specifications
(2) Functional correctness
(3) Verification-only specifications
(4) Proof assistance
(5) (Pre)state (or assumptions)
(6) Assumptions
(7) Behavioural specifiers
(8) Code
(9) Modelling
(10) Permissions
(11) Predicates
(12) Ghost state
(13) Other ghost stuff
(14) Exceptions
(15) Non-null
(16) Equal
(17) Arrays and iteration
(18) Does not fit in other groups

Due to the limitation of using 30 cards per study, whereas there are many different types of annotations (either in complexity, purpose or underlying logic), one may notice that some of the final categories in the taxonomy (see Section 3) have not been suggested by participants. Instead, the list of suggested categories has been used as a starting point for the taxonomy.

## 2.3 From card sorting results to taxonomy categories

We decided to use the following proposed top-level categories, because they all give a high-level description of what an annotation is used for. For each category we mention its relation to the suggested categories as mentioned in the previous section.

- Proof assistance (suggested as 4, also includes 6, 12, 13)
- Permissions (suggested as 10)
- Behavioural specification (suggested as 1 and 2)
- Functions (includes 11)
- Usability keywords (includes 7)
- Tool-specific specification (18)

Most of the other proposed categories were more detailed categories, therefore they have been included as subcategories. For example, the *exceptions*, *equal* and *non-null* have been included as subcategories of "Behavioural specifications" because they describe specific classes of behavioural specifications. *Assumptions*, *Ghost state* and *Other ghost stuff* have been included as subcategories of "Proof assistance" because they are all approaches that are used to provide extra information to the underlying solver. *Arrays and iteration* has not been included in the taxonomy because we decided to use categories that describe the purpose of an annotation whereas this category takes a data structure approach. However, the taxonomy includes several "Complexity" questions, such as "Does the annotation use quantifiers?" which can be used to gain some insight into the data structure that an annotation is about.

Other things that were considered when choosing subcategories for the taxonomy include:

- Frame conditions, which are annotations that state that a variable's value does not change, are "rarely written by developers in practice" [28] but are a promising case for annotation generation.
- Reasoning about termination typically requires additional annotations such as loop variants.
- We have separated functional specifications according to whether they are expressed as a "bound check" (<, >, <=, >=) or a "value check" (==, ! =, suggested as *equal*).

- "Value check" has been split into more subcategories as it contained many different annotations. These subcategories have been chosen based on what kind of variables are compared: whether they refer to a previous state of the program, whether they compare a variable to a hardcoded number, etc. "Bound check" has been split into subcategories as well.

## 3 TAXONOMY

In this section we present the taxonomy for auto-active verification annotations. As mentioned above, each annotation is categorised into one of the following top-level categories:

- *Proof assistance*: Annotations necessary to help the underlying solver to verify the program. It typically requires significant insight from the user to write such annotations.
- *Permissions*: Annotations used to describe access to certain variables, e.g. if a function may read from a certain variable or write to it.
- *Behavioural specification*: Annotations that describes the behaviour of the program.
- *Functions*: Annotations that can be used as functions in other annotations. This includes function definitions, predicate definitions (functions with a specific return type) or predicate usages in preconditions/postconditions/etc. This does *not* include the (un)folding of predicates or other function usages.
- *Usability keywords*: Annotations that make it easier to write a specification.
- *Tool-specific specifications*: Annotations specific to one of the tools.

In the rest of this section, we will discuss each of the above mentioned categories and their subcategories in detail. We also observed that there are annotation generators that cannot generate more complicated annotations such as annotations with quantifiers. To be able to evaluate these tools in more detail, it is important to know about the complexity of annotations. Therefore, we propose a set of questions about the complexity of an annotation in Section 3.7.

### 3.1 Proof assistance

Proof assistance annotations are used to provide extra information to the underlying solver. Each proof assistance annotation is classified according to the following subcategorisation:

- Assumption
- Axiom
- Ghost code
  - Definition of ghost variable
  - Setting/modifying the value of a ghost variable
  - Passing or receiving a ghost parameter
  - Other ghost code
- Lemma
- (Un)folding of a predicate

These subcategories have been chosen based on the suggested categories *Assumptions, Ghost state, Other ghost stuff* during the card sort. In addition, we also considered existing features in KeY, Krakatoa, OpenJML, VerCors, and Verifast.

An example of a proof assistance annotation is `axiom max_is_ge: \forall integer x y; max(x,y) >= x && max(x,y) >= y;`. This would be classified as *Proof assistance* ⟶ *Axiom.*

### 3.2 Permissions

Examples of "Permissions" are statements such as `assignable var`, `requires Perm(x, write)` or `var |-> _`. These annotations describe whether a method may read from or write to a variable. Many of the permission annotations can be found in examples from VerCors and Verifast because these tools use separation logic. Each permission annotation is further categorised according to their position in the program:

- Method: Used for annotations about a whole method, such as the annotations with the keywords `assignable`, `assigns`, `accessible`, `pure`, `modifiable`, and `modifies`.
- Precondition
- Postcondition
- Loop invariant
- Tool-specific: For positions that are specific to the technique of one tool.

It is important to note that permissions will be categorised as a predicate, instead of a permission, if they are used *in* a predicate. This is because predicates cannot be split into smaller equivalent chunks.

An example of a permission annotation is `requires (\forall* int i; 0<=i && i<a.length; Perm(a[i], write));`. This annotation would be categorised as *Permissions* ⟶ *Precondition.*

### 3.3 Behavioural specifications

Annotations about the behaviour of a program are classified in two ways. Firstly, the position in the code where they are found, and secondly the behaviour that they describe.

*3.3.1 Position in code.* The following positions have been included:

- Assertion/refute
- Precondition
- Postcondition
- Exceptional postcondition
- Loop invariant
- (Class) invariant
- Method (this includes annotations about a method as a whole).

*3.3.2 Behaviour.* Next, each annotation is classified according to the behaviour that it describes as follows:

- Termination (gives information about the termination of the code block)
  - Loops (gives information about the termination of a loop).
  - Methods (gives information about the termination of a method).
- Functional (gives information about the value of variables)
  - Bound check (when variables are compared with >, >=, <=, or <.)
    - Specific value (when a variable is compared to a hardcoded number, e.g. `a > 0`)
    - Related to other variable (when a variable is compared to another variable, e.g. `a > b`)

        * Related to previous state (when a variable is compared to its value in a previous state, e.g. `a > \old(a)`)
    - Value check (when variables are compared with == or !=).
        * Specific value
        * Related to other variable
        * Related to previous state
        * Null check (when a variable is compared to `null`, e.g. `a != null`)
        * Frame condition (when a variable's value stays the same, e.g. `a == \old(a)`)
        * Boolean check (when a boolean function is called or a comparison to a boolean is made)
        * Class check (when a variable's class is compared, indicated with `instanceof` or `a.getClass() == B.class`)
        * Default (no comparison with a variable, condition is either `true` or `false`).

An example of a behavioural specification is `signals_only NullPointerException` (equivalent to `signals (Exception e) e instanceof NullPointerException`). It checks whether the thrown exception is of the specified type `NullPointerException`. Therefore, this annotation would be categorised as *Behavioural specification → Exceptional postcondition & Functional → value check → class check.*

## 3.4 Functions

"Functions" is used for annotations that can be used as functions in other annotations. This does not include proof assistance annotations that can be written as a function such as axioms. It is used for function definitions, predicate definitions (which are function definitions with a specific return type) and predicate usages. Function usages are included in the "behavioural specification" category. Function annotations are categorised as follows:

- Predicate (indicated with the `predicate` keyword in Krakatoa and Verifast, and with the `resource` type in VerCors)
    - Definition of a predicate
    - Usage of a predicate
        * Precondition
        * Postcondition
        * Loop invariant
        * Assertion
- Function definitions (any function definition that is not a predicate definition)

An example of a predicate definition is `predicate valid_id(File child;) = [_]child.fileID |->_;`. This would be classified as *Function → Predicate → Definition of a predicate.* This predicate can be used in a postcondition (`ensures [f]valid_id(this);`) which would be classified as *Function → Predicate → Usage of a predicate → Postcondition.* A function definition can be something like `int max(int x,int y) = x > y ? x : y;` which returns the maximum of two integers. This is classified as *Function → Function definition.*

## 3.5 Usability keywords

"Usability keywords" are used for annotations that make it easier for the user to express certain constraints or specifications. These annotations do not describe functional properties of the program, i.e. they do not state anything about the value of variables. This category includes keywords such as `behaviour`, `normal_behaviour`, `exceptional_behaviour`, `also`, `helper` and `nullable`. These are keywords that may affect the verification, though most programs can be rewritten to express the same behaviour without these keywords.

For example, `behaviour` indicates a specification case. While they improve readability for users, they are not required and a specification with `behaviour` keywords can be rewritten into one without them.

## 3.6 Tool-specific

This category is used for annotations that do not fit into the general taxonomy as described above. Typically, these are statements that are used by one specific tool. We have encountered the following annotations in this category:

- `History` and `Future` related annotations (from the VerCors tool)
- `leak`, `init_class()`, `truncating`, `produce_lemma_function_pointer_c` `produce_call_below_perm_()`, fixpoints and inductive data types (from the Verifast tool)
- `\inv`, `represents` clauses (from the KeY tool)
- `model` variables and an `old` clause used for abbreviation of a specification case (from OpenJML)
- `logic` statements without definitions (from Krakatoa)

This category can be further extended in the future if other tool-specific annotations are encountered.

## 3.7 Complexity

Finally, we propose a set of questions about the complexity of an annotation. This can be used to evaluate existing annotation generators as well as for the teaching of auto-active verification. For example, if quantifiers are used very often, then this will be an important topic to cover. We will answer each of the following questions for an annotation:

- Does it include quantifiers?
- Does it include nested quantifiers?
- Does it include implications?
- Does it include a double implication?
- Does it include an inline if-statement?
- Does it include non-linear math (division, multiplication, modulo, etc.)?
- Does it include `\old`? (or uses patterns to bind the value in the precondition to a variable name in Verifast)
- Does it include `\result`? (or the Verifast equivalent `result` keyword)
- Does it include built-in language or verifier constructs? (e.g. `.length` to get the length of an array or `|seq|` to get the length of a sequence in VerCors)
- Does it include other method/function/axiom/lemma calls?

This is not an exhaustive list of statements that describe the complexity of an annotation. We have chosen these because we think that they are useful to evaluate the effectiveness of annotation generators.

**Table 1: The steps that have been taken to verify a program per tool.**

| Tool | Steps taken to verify a program |
|---|---|
| KeY | `java -jar key.jar`, then load `[file.java]` and for each contract target click "Start/stop automated search proof" until all proof goals have been proven. |
| Krakatoa | `krakatoa [file.java]` then apply the "Auto level 0" strategy to "All goals". |
| OpenJML | `java -jar openjml.jar -esc -progress [file.java]` |
| VerCors | `vct -silicon [file.java]`. The `–check-history` flag was also used for `LFQHist.java` and `NoSendAfterRead.java`. |
| Verifast | `./bin/vfide [file.java]` then press the "Verify" button. The `-disable_overflow_check` flag was used for `Contrib.java` and `client.java` as in Verifast's own test suite. |

## 4 ANALYSIS OF ANNOTATIONS IN JAVA PROGRAMS

In the previous sections we have shown how we set up a taxonomy for annotations used by auto-active verifiers. In this section we explain how we have used the taxonomy to classify a large set of verified Java programs.

### 4.1 Methodology

*4.1.1 Choosing samples.* For the analysis we have used the same randomly selected programs as for card sorting. These programs were randomly selected from the examples that were publicly available for each tool. For card sorting only 30 annotations were used, whereas we use all 1511 annotations in this analysis. We only included samples that are written in Java and can be verified automatically with KeY, Krakatoa, OpenJML, VerCors or Verifast.

Many of the randomly selected samples are relatively small programs. Therefore, to get a more representative data set, we have also included a larger case study for each tool except OpenJML. We have not included a case study for OpenJML since existing case studies are not publicly available. The following case studies have been included:

- KeY: a simplified implementation of a keyserver [9]
- Krakatoa: a genetic algorithm [7]
- VerCors: a red-black tree data structure [2]
- Verifast: Java Card API [25]

All of the random samples and the case studies were verified with the corresponding tool. Table 1 shows the steps that were taken to verify each sample.

*4.1.2 Data cleaning.* After selecting samples, the annotations in the chosen files needed to be extracted. We only extracted verifiable annotations. If only a part of the file could be verified automatically, then only those annotations have been included.

Next, the annotations were desugared, i.e. removing syntactic sugar, which resulted in smaller statements that are logically equivalent for most of the verifiers[6]. The annotations were desugared according to the rules described in Table 2. Similar rules to the ones for preconditions (annotations that start with `requires`) apply to postconditions, loop invariants and assertions.

After each annotation was manually extracted and desugared, it was categorised according to the taxonomy (as presented in Section 3). For quantified statements the quantified part is categorised, not the ranges of the quantified variable. If an annotation still contained multiple logical statements after the data cleaning, e.g. $a||b$, then this statement is classified for $a$ and for $b$. If $a$ and $b$ are classified the same, then this results in one categorisation. Otherwise, the annotation receives multiple categorisations.

For example, `loop_invariant x > 0 ==> height > 0;` contains the logical statements `x > 0` and `height > 0`. Both of these are categorised as *Functional → Bound check → Specific value*. Therefore, the statement is also categorised as *Functional → Bound check → Specific value*. However, `invariant my_inv: balance >= 0 && balance <= MAX_BALANCE;` would be categorised as *Functional → Bound check → Specific value* and *Functional → Bound check → Related to other variable*.

### 4.2 Results

Next, we report how often the annotation types of the taxonomy occur in our data set. Specifically, we report on the usage of the top-level categories and the subcategories of the two most common top-level categories namely behavioural subcategories and proof assistance subcategories. Moreover, we report on the complexity of annotations as this is a special category in the taxonomy. The data set, including the categorised annotations, is freely available at **https://doi.org/10.4121/16545714** for inspecting additional details or for the reader's further research interests.

In addition, we also report a 95% confidence interval (CI) for the annotation types that are mentioned. The 95% CI indicates how many of that annotation type (in percentages) can be expected in a Java program that is verified with either KeY, Krakatoa, OpenJML, VerCors or Verifast. It can be interpreted as follows: If one were to repeat our experiment multiple times, then 95% of the calculated confidence intervals (which would differ each time) would encompass the true mean, namely the percentage that indicates how often that annotation type occurs in a Java file that has been verified with one of the five verifiers.

The 95% CIs have been calculated based on the annotations per program (50 samples + 4 case studies). Each case study has been included as one data point, so we can reasonably assume that all programs are mutually independent. To calculate the CIs, we use a t-distribution to calculate the expected mean because we have an unknown population mean and variance.

The complete data set consists of 10k+ lines, of which 3k+ code, almost 5k lines of annotation, 1k empty lines and 1,5k comment lines. From this we have extracted 4610 (desugared) annotations. This means that, on average, 1,34 annotations per line of code are written. Note that this looks at the number of annotations instead

[6]All verifiers except Verifast support multiple pre-/postconditions per method. These are equivalent to the conjunction of the multiple statements.

**Table 2: This table shows the rules that were applied to split annotations into smaller logically equivalent annotations. The left column shows the original annotation and right column shows how this annotation has been split. &*& and ** denote the separating conjunction [26].**

| Original annotation | Split into |
|---|---|
| `requires a && b` | `requires a` and `requires b` |
| `requires a &*& b` or `requires a ** b` | `requires a` and `requires b` |
| `requires (\forall int i; ..<i && i<...; a && b);` | `requires (\forall int i; ..<i && i<...; a);` and `requires (\forall int i; ..<i && i<...; b);` |
| a<b<c | a<b and b<c |
| `PointsTo(var, p, val)` | `Perm(var, p)` and `var == val` |
| `context a` | `requires a` and `ensures a` |
| `context_everywhere a` | `loop_invariant a` for each loop in the method, `requires a` and `ensures a` |
| `assignable a, b` | `assignable a` and `assignable b` |
| `a => b && c` | `a => b` and `a => c` |
| `(a ? b : c)` (at top-level) | `a => b` and `!a => c` |
| `\unfolding pred() in a ** b` | `\unfolding pred() in a` and `\unfolding pred() in b` |

of the number of annotation lines because one line can include multiple annotations and annotations can occupy multiple lines.

*4.2.1 Top-level categories.* An overview of how often the different top-level type annotations occur can be found in Table 3. The percentages are given in terms of the total number of annotations used by that tool in our data set, e.g. 11,2% of all annotations used by the KeY verifier are used for proof assistance.

We can conclude that behavioural specifications make up the largest part of annotations for all tools. For a Java program, that has been verified with one of the analysed tools, we can expect 47%-62% of the annotations to be behavioural specifications. Proof assistance annotations are expected to account for 9%-22% of all annotations.

*4.2.2 Position of predicates, behavioural and permission annotations.* Predicates (definitions and usages), behavioural and permission annotations have also been categorised according to their position in the code. Next, we have a look at the most common positions for these annotations (see Table 4). We can conclude that these annotations are expected to be found most often in preconditions (16%-22%), postconditions (18%-27%) and loop invariants (14%-28%). However, exceptional postconditions are expected to occur very little (0%-2%). Method annotations are most commonly used in programs verified with Key, Krakatoa and OpenJML. Verifast and VerCors rarely use method annotations.

*4.2.3 Behavioural specifications.* [7] Next, we discuss the usage of (subcategories of) behavioural specifications. Functional specifications (43%-58%) are significantly more common than annotations about the termination (1%-6%) of a code block. This is to be expected because many auto-active verifiers assume that a program terminates. The most common functional specifications are bound checks where a variable is compared to either a specific value (9%-17%), bound checks where a variable is compared to another variable (7%-15%) or value checks where a variable is compared to another variable (7%-14%).

[7]The results for these subcategories are not presented in separate tables. Instead, they can be inspected in more detail in the published data set [21].

*4.2.4 Proof assistance.* [7] In a Java program that has been verified with one of the five analysed tools, the most commonly expected proof assistance annotations are ghost code (2%-11%) and predicate (un)folds (2%-7%). KeY, Krakatoa and OpenJML examples never use predicate (un)folding in our data set. Therefore, we can expect the predicate (un)folds to make up a larger part of annotations for VerCors and Verifast examples than indicated by the CI.

*4.2.5 Complexity.* [7] Aside from the type of annotations that are used, we are also interested in the complexity of the annotation. For this we looked into the questions mentioned in Section 3.7 such as whether the annotation uses quantifiers or implications.

The constructs that we can expect most often in annotations of a verified Java program are built-ins (16%-31%), quantifiers (8%-18%) and calls to other methods/lemmas/etc. (5%-14%). This indicates that it is important for annotation generation techniques to consider calls to other methods, lemmas, etc. as well as the usage of built-in constructs of Java or the verifier. Moreover, techniques that do not support quantifiers are expected to be unable to generate 8%-18% of the required annotations for a Java program.

## 4.3 Reflecting on the results

To reflect on the results from Section 4.2, we have interviewed six experts to learn about the annotations that they use for deductive verification. We asked questions about the types of annotations that they use, which annotations they used the most and the least, and we asked for their reaction on our results presented in Section 4.2. We used the interviews to determine whether our results confirm existing beliefs or whether some findings are unexpected.

We have interviewed David Cok, Bart Jacobs, Jean-Christophe Fillîatre, Mattias Ulbrich, Nikolai Kosmatov and Wolfgang Ahrendt. They have experience with different verifiers including Frama-C, KeY, OpenJML, Verifast and Why3. The type of programs that they typically verify span a wide range, including distributed programs, concurrent programs, examples used in education, data structures,

**Table 3: Overview of how often the top-level taxonomy types occur in the data set.**

| | KeY | Krakatoa | OpenJML | VerCors | Verifast | Mean | 95% CI |
|---|---|---|---|---|---|---|---|
| Proof assistance | 11,2% | 5,5% | 7,0% | 24,9% | 29,7% | 23,7% | 9-22% |
| Permissions | 15,5% | 10,8% | 5,6% | 10,8% | 10,2% | 10,8% | 9-17% |
| Behavioural | 60,0% | 61,8% | 78,2% | 46,8% | 23,5% | 41,1% | 47-62% |
| Functions | 0% | 18,8% | 0% | 16,4% | 37,6% | 22,7% | 5-15% |
| Usability keywords | 11,7% | 2,5% | 6,3% | 0% | 0% | 1,3% | 2-8% |
| Tool-specific | 1,6% | 0,6% | 2,8% | 2,0% | 1,2% | 1,6% | 1-4% |

**Table 4: Overview of the positions of behavioural, predicates (definitions and usages) and permission annotations in the code.**

| | KeY | Krakatoa | OpenJML | VerCors | Verifast | Mean | 95% CI |
|---|---|---|---|---|---|---|---|
| Method | 13,1% | 10,8% | 5,6% | 0,1% | 0,7% | 2,3% | 4-9% |
| Assertion | 0% | 11,1% | 0% | 2,7% | 2,8% | 3,0% | 1-4% |
| (Class) invariant | 8,3% | 1,5% | 0,7% | 0% | 0% | 0,8% | 1-3% |
| Loop invariant | 16,5% | 31,7% | 43,7% | 11,0% | 17,0% | 16,2% | 14-28% |
| Preconditions | 15,7% | 14,2% | 16,9% | 25,3% | 21,5% | 22,1% | 16-22% |
| Postconditions | 21,3% | 17,8% | 14,8% | 29,3% | 23,4% | 25,2% | 18-27% |
| Exceptional postconditions | 0,5% | 0,3% | 2,1% | 0% | 0% | 0,1% | 0-2% |
| Predicate definitions | 0,0% | 3,1% | 0% | 1,5% | 3,6% | 2,2% | 1-5% |

complex algorithms, security related programs and industrial case studies.

*4.3.1 Discrepancies.* First, we will discuss three points that interviewees pointed out that were not reflected by the data.

Two interviewees mentioned that (class) invariants are annotations that they use often. This is not reflected by our results where (class) invariants are expected to make up only between 1-3% of all annotations. One of the reasons for this discrepancy is that VerCors and Verifast do not use invariants. As a result, the expected number of invariants is on average lower than one might expect for tools that do support invariants. One expert also mentioned that the use of invariants depends on the program that you are verifying. It is possible that our data set happens to include mostly samples that did not use invariants. This may be because we included more small verified programs as opposed to case studies.

An interviewee mentioned that the expected amount of ghost code seemed high (2-11%). If you use ghost code, then you likely have multiple ghost code statements. So either you use no ghost code, or you probably have multiple ghost code statements. This would also explain the large expected range, whereas other proof assistance annotations tend to have a confidence interval range of around 4% or lower.

Several experts pointed out that the required annotations highly depend on the program that you are verifying. While this is true, this research aims to show what annotations a user would need to write on average to verify a Java program with one of the tools.

*4.3.2 Confirmed by data.* Some specific points in the data that were confirmed by multiple interviewees include:

- Preconditions, postconditions and loop invariants are used a lot.
- Exceptional postconditions are used rarely.
- Assumptions and axioms occur little.
- Termination-related annotations are used rarely.
- A typical loop invariant is `0 < i < N`. Our results show that bound checks where a variable is compared to a specific value (0 < i) and to another variable (i < N) are indeed one of the most commonly expected functional specifications.

Based on these interviews, our results seem to be in line with the expectations of professionals. The most important thing to keep in mind, is that the confidence interval may give a skewed view in case some annotations are only used by a subset of the tools. This is especially important for tool builders who target a specific verifier.

## 4.4 Using the results

Many annotation generators are evaluated to show how well they work on a subset of annotations instead of the impact on the overall process. Based on our results, we can evaluate the impact of these tools on the overall verification process without needing to do an extensive evaluation for each tool separately. In this section we show how to use the results of Section 4.2 to achieve this.

We have selected seven tools that generate annotations and identified the types of annotations that they infer. We identified in which taxonomy category the annotations belong. If there was no category that exactly matches the type of annotations that are inferred, then we selected a category that included all possible annotations that could be inferred. Next, we estimated their impact based on the taxonomy category and the corresponding confidence intervals (see Section 4.2). The results are presented in Table 5.

As an example, we discuss one tool, DynaMate, to show how to interpret the results. DynaMate generates loop invariants. The corresponding taxonomy category is the “Loop invariant" position

**Table 5: An evaluation of the impact of annotation generators on the overall verification process. Based on the type of annotations that the tool generates, we find a corresponding category in the taxonomy and the expected impact based on the results of this research (see Section 4.2).**

| Tool name | Type of annotations inferred | Corresponding taxonomy category | Estimated maximum impact |
|---|---|---|---|
| C2S [33] | Preconditions, normal postconditions and exceptional postconditions | Preconditions | 16-22% |
| | | (Normal) postconditions | 18-27% |
| | | Exceptional postconditions | 0-2% |
| DynaMate [15] | Loop invariants | Loop invariants | 14-28% |
| Sample [11] | Permission pre- and postconditions for array programs | Permissions (all subcategories included) | 9-17% |
| ShaPE [6] | Recursive shape predicates | Functions → Predicates → Definition of a predicate | 0-5% |
| SLING [22] | Preconditions, postconditions and loop invariants for heap-manipulating programs | Preconditions | 16-22% |
| | | Postconditions | 18-27% |
| | | Loop invariants | 14-28% |
| Strongarm [29] | Postconditions | Postconditions | 18-27% |
| Verifast [31] | Auto open/close statements, lemma applications, postconditions and loop invariants | Open/close predicates | 0-3% |
| | | Lemmas | 0-4% |
| | | Loop invariants | 14-28% |
| | | Postconditions | 18-27% |

category for behavioural specifications, predicate usages and permissions (see Table 4). Loop invariants are expected to make up between 14-28% of all required annotations. Therefore, DynaMate is expected to be able to generate at most 14-28% of all required annotations.

Based on Table 5, we can identify areas that would be interesting for future research:

- Proof assistance: Only a few proof assistance annotations can be generated right now (lemmas and opening/closing of predicates) even though they are expected to make up a significant (9-22%) part of all required annotations.
- Usability keywords: It is important to generate annotations that are easy to understand for the user so the user can still check whether they express the desired behaviour. The annotation generators from Table 5 do not generate annotations that use keywords such as `behaviour`.
- Permissions: Of the mentioned tools, only one tool generates permissions. Since permissions are expected to account for 9-17% of all required annotations, it would be worthwhile to further investigate the inference of permission annotations.

## 5 DISCUSSION

In this section, we discuss some limitations of the card sorting, taxonomy and the analysis of annotations in Java programs, as discussed in this paper.

The *scope* of this research has been limited to Java programs. Therefore, our conclusions do not necessarily reflect what auto-active verifiers for other languages would need. For example, it could be that a verifier for a dynamically-typed language would need information about the type of a variable. In such cases, the taxonomy can be extended to include new types of specifications. Note that we publish the data set of all analysed annotations [21] to make it easy to extend the analysis to include other verifiers.

For the card sorting and analysis, we *selected samples* that had already been verified with the chosen tools. Another way to choose samples would have been to take a set of programs and write annotations so that each tool could verify the same programs. One could then try to choose a set of programs that would form a representative set of Java programs. However, different tools use different techniques and support different parts of the Java language, making it difficult and time consuming to verify several programs for each tool. By choosing examples from the tools themselves, we avoid this problem, while still reflecting the most commonly used annotations. As a result, some programming constructs that are difficult to verify, such as inheritance and exceptions, are underrepresented in our sample set. This means that our chosen samples are not a representative set for real-world Java code. Instead, the samples reflect what the verifiers are currently (at least) capable of verifying.

The card sorting participants all have *experience* with *VerCors*. The proposed taxonomy can therefore be biased towards VerCors. To minimise the impact of this bias, we were careful to avoid any categories in the taxonomy that would only apply to VerCors. It inspired us to create the category called “Tool-specific".

Some of the annotations were *categorised twice*. Therefore, some of the numbers in the analysis might not seem to add up, e.g. the Verifast case study had 24,81% functional specifications, of which 3,00% bound checks and 21,98% value checks. This is caused by 2 annotations that are categorised as both bound checks and value checks. These annotations used `||` which cannot be split into smaller equivalent statements. As explained, such an annotation receives multiple categorisations, one for each type of statement that occurs. This impacts only a small part of the data set and the results still accurately reflect how often each subtype occurs.

Finally, we assume that the annotations are a *minimal set* needed to prove correctness. We did not try to minimise the written annotations and we did not check whether the specifications are complete. Therefore, depending on what you want to prove, it could be that more annotations are needed. Related to this, we also note that the Krakatoa case study had 3 goals that could not be proven automatically. These included the statements `\result <= attackStrength_logic(intensity)`, `System.out != null` and `\forall i; 0 <= 0 && i <array_index; states[i] != this`. These unproven proof obligations have been mentioned in [7]. We have still included the case study as it was the only available Krakatoa case study that we could find and the majority could be proven. To be able to prove these statements, one would likely need additional annotations. We did not remove these unproven statements in case the proofs for other annotations depend on these. As this affects only a small part of the data set, we can still derive meaningful results such as the most common types of annotations.

## 6 RELATED WORK

Several researchers have analysed how specifications are used. However, most focus on specifications used for lightweight approaches such as debugging, instead of auto-active verification.

For example, Estler et al. [12] have analysed contracts which are expressed as executable assertions in the form of pre-/postconditions and class invariants. They evaluated which contract elements are used most often and how contracts change over time. Unlike Estler et al. our research takes a more detailed look at specifications that are used for auto-active verification. Like us, Estler et al. have measured how often `\old` is used in postconditions (2%-3%) as well as null checks and quantification. In our data set, the number of annotations using `\old` for a verified Java program is a lot higher (4%-10%). This difference may arise from the fact that Estler et al. only include postconditions whereas we investigate all annotations which include `\old`.

Schiller et al. [28] studied the usage of Microsoft Code Contracts, specifically preconditions, postconditions and invariants. Only 1 of the 4 projects analysed by Schiller et al. uses static checking. The other projects have specifications that are used by lightweight techniques such as debugging and runtime error detection. They found a large number of nullness (e.g. `a != null`) contracts (22%-33% of inferred specifications and 55%-69% of developer written specifications) whereas this is expected to only make up between 1,83%-4,77% of annotations in a verified Java program according to our research. However, Schiller et al. compare what is written by developers to what is inferred by the Daikon invariant detector, whereas our research looks at all annotations used by an auto-active verifier.

Dietrich et al. [10] have performed an empirical study of techniques and tools for lightweight contract checking in Java. Similar to Estler et al., they investigated contracts used as lightweight specifications, as well as how they change over time. And, similar to the study by Schiller et al, they focused on preconditions, postconditions and class invariants. While they did categorise annotations, we present a more detailed categorisation in this work. Moreover, we focus on annotations for auto-active verification as opposed to lightweight contract checking.

Unlike the approaches mentioned above, Furia et al. [14] have looked at programs that have been verified with deductive verification. They analysed loop invariant patterns that are used in several algorithms, and used this to present a classification for loop invariants. Our research presents a continuation of their work by presenting a classification for all annotations for auto-active verifiers. Furia et al. have classified loop invariants based on their role and the transformation that yields the invariant from the postcondition. We, however, categorise annotations independently of their context.

## 7 CONCLUSION AND FUTURE WORK

This paper presents the first detailed taxonomy for categorising all annotations used by auto-active verifiers for Java programs. Based on this taxonomy, we have published and extensively analysed a data set of annotations from verified Java programs. In this analysis we have included five different auto-active verifiers, KeY, Krakatoa, OpenJML, VerCors and Verifast. We have included both smaller programs and larger case studies. The analysis shows what annotations can be expected in verified Java programs. For example, annotations that describe the behaviour of the program are expected to make up 47%-62% of all annotations of a verified Java program. Furthermore, approximately 8%-18% of required annotations for a Java program are expected to include a quantifier. Based on interviews, we conclude that the results seem to be in line with the expectations of experts. These results are an invaluable resource to evaluate the impact of annotation generators, as we have shown in Section 4.4. Moreover, it gives insight into the current state of the art of auto-active verifiers, such as which annotations are most frequently used and are therefore important to teach to new verification engineers.

In future work, the data set can be expanded with examples from other verifiers, e.g. Dafny, to see whether the results are similar. Moreover, the taxonomy can be extended for other languages.

## ACKNOWLEDGMENTS

We are very grateful for the experts that participated in the interviews: Wolfgang Ahrendt, David Cok, Jean-Christophe Filliâtre, Bart Jacobs, Nikolai Kosmatov and Mattias Ulbrich. This research was supported by the NWO VICI 639.023.710 Mercedes project.

## REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). 2016. *Deductive Software Verification - The KeY Book - From Theory to Practice.* Lecture Notes in Computer Science, Vol. 10001. Springer. https://doi.org/10.1007/978-3-319-49812-6 Tool website: https://www.key-project.org/.

[2] Lukas Armborst and Marieke Huisman. 2021. Permission-Based Verification of Red-Black Trees and Their Merging. In *9th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2021, Madrid, Spain, May 17-21, 2021.* IEEE, 111–123. https://doi.org/10.1109/FormaliSE52586.2021.00017

[3] Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. 2012. Lessons Learned From Microkernel Verification – Specification is the New Bottleneck. In *Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012 (EPTCS, Vol. 102),* Franck Cassez, Ralf Huuck, Gerwin Klein, and Bastian Schlich (Eds.). 18–32. https://doi.org/10.4204/EPTCS.102.4

[4] Bernhard Beckert and Reiner Hähnle. 2014. Reasoning and Verification: State of the Art and Current Trends. *IEEE Intell. Syst.* 29, 1 (2014), 20–29. https://doi.org/10.1109/MIS.2014.3

[5] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 2017. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *Integrated*

*Formal Methods - 13th International Conference, IFM 2017, Turin, Italy, September 20-22, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10510).* Springer, 102–110. https://doi.org/10.1007/978-3-319-66845-1_7 Code: https://github.com/utwente-fmt/vercors/, commit: 0aa6a9a5fba5279f69e8f8ccce5173a70b558ed3.

[6] Jan H. Boockmann and Gerald Lüttgen. 2020. Learning Data Structure Shapes from Memory Graphs. In *LPAR 2020: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Alicante, Spain, May 22-27, 2020 (EPiC Series in Computing, Vol. 73).* EasyChair, 151–168. https://easychair.org/publications/paper/mkjl

[7] Dmitry Brizhinev and Rajeev Goré. 2018. A case study in formal verification of a Java program. https://arxiv.org/abs/1809.03162.

[8] David R. Cok. 2011. OpenJML: JML for Java 7 by Extending OpenJDK. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617).* Springer, 472–479. https://doi.org/10.1007/978-3-642-20398-5_35 Tool website: https://www.openjml.org/.

[9] Stijn de Gouw, Mattias Ulbrich, and Alexander Weigl. 2020. verifythis-ltc-2020. https://github.com/KeYProject/verifythis-ltc-2020.

[10] Jens Dietrich, David J. Pearce, Kamil Jezek, and Premek Brada. 2017. Contracts in the Wild: A Study of Java Programs. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74).* Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:29. https://doi.org/10.4230/LIPIcs.ECOOP.2017.9

[11] Jérôme Dohrau, Alexander J. Summers, Caterina Urban, Severin Münger, and Peter Müller. 2018. Permission Inference for Array Programs. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10982).* Springer, 55–74. https://doi.org/10.1007/978-3-319-96142-2_7

[12] H.-Christian Estler, Carlo A. Furia, Martin Nordio, Marco Piccioni, and Bertrand Meyer. 2014. Contracts in Practice. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8442).* Springer, 230–246. https://doi.org/10.1007/978-3-319-06410-9_17

[13] Jean-Christophe Filliâtre and Claude Marché. 2007. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4590).* Springer, 173–177. https://doi.org/10.1007/978-3-540-73368-3_21

[14] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: Analysis, classification, and examples. *ACM Compututing Surveys (CSUR)* 46, 3 (2014), 34:1–34:51. https://doi.org/10.1145/2506375

[15] Juan Pablo Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2014. DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification. In *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).* Springer, 48–53. https://doi.org/10.1007/978-3-319-13338-6_4

[16] Dilian Gurov, Christian Lidström, Mattias Nyberg, and Jonas Westman. 2017. Deductive Functional Verification of Safety-Critical Embedded C-Code: An Experience Report. In *Critical Systems: Formal Methods and Automated Verification - Joint 22nd International Workshop on Formal Methods for Industrial Critical Systems - and - 17th International Workshop on Automated Verification of Critical Systems, FMICS-AVoCS 2017, Turin, Italy, September 18-20, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10471).* Springer, 3–18. https://doi.org/10.1007/978-3-319-67113-0_1

[17] Reiner Hähnle and Marieke Huisman. 2019. Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools. In *Computing and Software Science - State of the Art and Perspectives.* Lecture Notes in Computer Science, Vol. 10000. Springer, 345–373. https://doi.org/10.1007/978-3-319-91908-9_18

[18] Marieke Huisman and Raúl E. Monti. 2020. On the Industrial Application of Critical Software Verification with VerCors. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 12478).* Springer, 273–292. https://doi.org/10.1007/978-3-030-61467-6_18

[19] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 2011. VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617).* Springer, 41–55. https://doi.org/10.1007/978-3-642-20398-5_4 Code: https://github.com/verifast/verifast.

[20] Alexander Knüppel, Thomas Thüm, Carsten Pardylla, and Ina Schaefer. 2018. Experience Report on Formally Verifying Parts of OpenJDK's API with KeY. In *Proceedings 4th Workshop on Formal Integrated Development Environment, F-IDE@FLoC 2018, Oxford, England, 14 July 2018 (EPTCS, Vol. 284).* 53–70. https://doi.org/10.4204/EPTCS.284.5

[21] Sophie Lathouwers and Marieke Huisman. 2022. Database of Annotations for Deductive Verifiers. https://doi.org/10.4121/16545714 Accessed on: 28 March 2022.

[22] Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. 2019. SLING: using dynamic analysis to infer program invariants in separation logic. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.* ACM, 788–801. https://doi.org/10.1145/3314221.3314634

[23] K. Rustan M. Leino and Michał Moskal. 2010. Usable auto-active verification. In *Usable Verification Workshop.* Citeseer. http://fm.csl.sri.com/UV10/

[24] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. 2004. The KRAKATOA tool for certificationof JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming* 58, 1-2 (2004), 89–106. https://doi.org/10.1016/j.jlap.2003.07.006 Tool website: http://krakatoa.lri.fr/.

[25] Pieter Philippaerts, Frédéric Vogels, Jan Smans, Bart Jacobs, and Frank Piessens. 2011. The Belgian Electronic Identity Card: a Verification Case Study. In *Proceedings of the International Workshop Automated Verification of Critical Systems (AVOCS'11)*, Vol. 46. https://doi.org/10.14279/tuj.eceasst.46.682

[26] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings.* IEEE Computer Society, 55–74. https://doi.org/10.1109/LICS.2002.1029817

[27] Christoph Scheben. 2014. *Program-level Specification and Deductive Verification of Security Properties.* Ph. D. Dissertation. Karlsruhe Institute of Technology. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046878

[28] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. 2014. Case studies and tools for contract specifications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014.* ACM, 596–607. https://doi.org/10.1145/2568225.2568285

[29] John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David R. Cok. 2019. Inferring Concise Specifications of APIs. (2019). arXiv:1905.06847 http://arxiv.org/abs/1905.06847

[30] Donna Spencer. 2009. *Card sorting: Designing usable categories.* Rosenfeld Media.

[31] Frédéric Vogels, Bart Jacobs, Frank Piessens, and Jan Smans. 2011. Annotation Inference for Separation Logic Based Verifiers. In *Formal Techniques for Distributed Systems - Joint 13th IFIP WG 6.1 International Conference, FMOODS 2011, and 31st IFIP WG 6.1 International Conference, FORTE 2011, Reykjavik, Iceland, June 6-9, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6722).* Springer, 319–333. https://doi.org/10.1007/978-3-642-21461-5_21

[32] Jed R. Wood and Larry E. Wood. 2008. Card Sorting: Current Practices and Beyond. *Journal of Usability Studies* 4, 1 (Nov. 2008), 1–6.

[33] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. 2020. C2S: translating natural language comments to formal program specifications. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020.* ACM, 25–37. https://doi.org/10.1145/3368089.3409716