Automatic Generation of Test Cases for VerCors Verification Cases Failing with Null Errors

Matthias Sleurink
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
m.sleurink@student.utwente.nl

ABSTRACT

VerCors is a toolset for static verification of concurrent and sequential programs. However, the error messages it creates for verification failures are not always the easiest to understand. The code involved may not be completely visible, and context from other methods may be missing completely. We have created a system to automatically generate a test case that displays how the error that VerCors finds can occur. Said system performs this task for the small but frequent subset of errors that can occur, the so-called "*nullpointer errors*". These occur when the code assumes that an object has a value, but in reality it does not (it is *null*). This generated test will allow more efficient use of the VerCors toolset for this error case.

Keywords

VerCors, Test Generation, Software Verification, Null Safety

1. INTRODUCTION

Software correctness and reliability is very important. Banking software for example, must make sure that money is correctly transferred, software in hospitals must register medicine intake accurately, and self driving cars must be able to stop in time. Failure from these or other software systems can have catastrophic results. We need a system that can contribute to guaranteeing reliability and finding mistakes.

Many such systems already exists, users of IDEs have almost surely seen the result of such systems. The IntelliJ IDEA[4] IDE gives the programmer warnings for when code cannot compile, it can warn the user when they use code constructs that have faster alternatives, and it can help the user investigate the nullity of variables.

Another example of such tools is VerCors. VerCors is a toolset that uses static verification to verify concurrent and sequential programs[3]. The error messages that VerCors generates can be hard to understand. The code involved may not be completely visible, and context from other places may be missing completely. We have added functionality to VerCors that generates a test case for certain cases of failing verification. This adds more context to the error messages that VerCors generates, and may help in the solving of these issues.

When the VerCors toolset reports an error it is reported to the log with some context. This context includes the file location, some surrounding lines, and an arrow that points to the *statement* that causes the fault. When this error is of the type *PostConditionFailed:AssertionFalse* or *AssertFailed:AssertionFalse* we know that this error is about an annotation that Viper could disprove.

Then our tool kicks in. We analyse the error and the file location that it occurs in. If the error is about a *non-null* annotation on the return value of the method we add a test case to the log. This test case can be seen as a counterexample to the annotation. We show the user how the annotation can be disproven by setting up the internal state of an object and calling the method in question with specific parameters to show how the method can be coerced to return *null*.

This generated test case can help the user in the investigation process, as it shows the user what state the program needs to be in for the issue to occur. The user can then either add more annotations to the method, for example an annotation to restrict one of the parameter values, or change the method code in such a way that the error does not occur anymore.

2. BACKGROUND

In the following section we will clarify background information that is used throughout this paper, as well as explain the context for which this tool was written.

2.1 Static Verification

Static verification is a technique where source code is analysed without being executed[12]. This technique is used by tools to check source code for different types of faults. From style concerns to performance issues to the correctness of concurrency constructs and more[3, 4, 6, 11]. Static analysis tools use their knowledge of a language to give advice or report errors to their users. VerCors is one of those tools.

2.2 VerCors

VerCors uses annotations to add claims to methods, these claims can be *preconditions*, where the user tells VerCors about the input, *postconditions*, where the user makes claims about the state after running a method, and other features like *loop invariants* and *permission* logic. Pre- and Post-conditions are used to tell VerCors things about variables and state. These annotations can range from limits on the value of variables, to *nullity*, to the length of arrays, and more.

35th Twente Student Conference on IT July. 2nd, 2021, Enschede, The Netherlands.
Copyright 2021, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

VerCors itself uses Viper[5] as a backend for verification. VerCors transforms the source input into a state where Viper can understand and analyse[3], and in turn interprets the messages coming back from Viper to show them to the user with some extra context.

Usage of the VerCors toolset is as follows. A user has a source file with a number of annotations. VerCors runs, and tells the user that it has found an *AssertionError* if VerCors has found that one of the claims made by the user is incorrect. Now it is on the user to find how this could occur. To help with this VerCors shows the user the *statements* involved in the error, and a number of lines of code that surround the statement.

This analysis process is where our tool comes in. If the user has written a postcondition like *//@ ensures \result != null;*, but VerCors finds that the method can in fact return *null* our tool shows a test case (or: counterexample) to show how this happens. We show the user a runnable piece of code that sets up the required values for instance variables and method parameters alike, and then calls the method. This can help the user to understand the error, and can thus help in the investigation process.

3. IMPLEMENTATION DESIGN

In this section we will explain the implementation design of our tool. We will first show an overview of what steps our tool takes from start to finish. For each of these steps we will then explain the interesting data structures and algorithms involved below.

As illustrating example we will show for every section what its code does with regards to the first example as shown in section 4.1. The input code for that example is as follows:

```
public class example_1 {
  //@ ensures \result != null;
  public int[] testing(int[] inp) {
    return inp;
  }
}
```

3.1 Overview

After VerCors has read the input, transformed it for checking by *Viper*[5], executed Viper on the input, and then interpreted the returned messages (see section 2.2 on page 1) our tool is executed by VerCors.

Our first step is to interpret the errors that VerCors has created from the Viper result. These errors contain a filename, a line and column number, as well as some error categorisation information. From this we check if this is an error that we can handle (*null* return errors).

If it is indeed a null return error we move along to step two. Here we find out what flow control statements have to be passed to gain access to the location of the error. When we have found all these constraints we move along to the next step.

The third step in the process is finding the variables that are involved. We extract method parameters and instance variables with constraints.

With these variable names in hand we move along to step four, where we utilise these constraints and names to figure out the value that these variables must have.

These required values are then converted into initialisers in step five. Which we then convert into *String* form to print to the output log.

With these initialisers we can set the value for the found required instance variables and method parameters in step six.

The seventh and last step is to use these method parameters to set up the code to call the method in question and print it to the log.

3.2 Interpreting Viper Errors

VerCors passes the input, after preparing it, to Viper. Viper in turn reports on any issues that it finds. VerCors then reads and interprets these issues, and passes them on to our tool.

We have a system in place where any tool can look at this error and check to see if they want to perform actions upon it. For now the only tool that is registered is ours. First we check if the error has two inner "error locations." If it has two or more, then the first should fall on a return statement. We check this by taking the location from the first error. This location contains a file path, that we read and parse with an Antlr4[8] parser, and a line and column index. We use these two numbers to find the *ParserRule* that is at that location in the file, if any. (More information about how we do that in the section 3.3.) If this ParserRule is a return statement then we know that this error *can* be the correct type for our tool to work on.

We continue with the second error location. To help users diagnose errors VerCors not only shows them the place where the error occurs, but also why that is an error. So in our case, where a method returns *null* but should not, the first error points to the return statement, and the second error points to an annotation. We use, once more, the line and column indexes to get the statement at this second location. If there is one we use a custom *tree-walker*-like algorithm to check if the annotation is of the following shape: "*//@ ensures \result != null;*"

The advantage of using an Antlr4 Java parser instead of reading the file and checking the characters ourselves lies in the large amount of different ways that a VerCors postcondition and a return statement can be written. We would have to keep in mind whitespace, (block) comments, multi-line statements, and more. Using the knowledge that the VerCors Java parsers already have to circumvent these issues is efficient from both an implementation robustness and development time standpoint.

In our example code Viper finds one error with two causes. The error is of type *PostConditionFailed:AssertionFalse.* Which is one of the types that we operate on. Then we find that the first location points to a return statement. We then look at the second error location. If the ParserRule at this location is one that describes an annotation we check to see if it is an annotation that describes the allowed value of the return value. If this is the case we check if it is compared with null, and if the comparator is "!=". When this is all true, and for the given example it is, then we know that this method has an annotation that tells us the return is not allowed to be null. And that Viper has found that it can be null. So our tool should generate a test for this input.

3.3 Find at Location

To find out if the error is one that we are interested in we need to check if certain locations of a java source file have a specific meaning. For that we use an *Antlr4* listener. This listener takes a line and a column index, and "listens" to every parsed rule to find the most specific ParserRule that is at the right location.

This system makes use of the handy feature that Antlr4 tree walkers have where they first visit a rule, and after that, visit its children. Due to this, when you have visited every rule, you can be sure that the last rule that matched the location is as "deep" or "specific" as possible. This makes it easier for users of this system to gather knowledge from the result, as they will not have to walk into several "wrapper" rules to find the one they are looking for.

This system is used several times in the given example. Once, for example, to check if the first error points to a return statement. In that case the finder receives the line number and column index that are stored in the Viper error. We store the ParserRule for which the line number and column index match the numbers that are in the Viper error and pass it on to the caller.

3.4 Constraints and Finding Them

Constraints are used for storing information about the limits that are placed on the values that variables may have. We store *source*s for variables, restrictions on their values, and a note that says if we want the test to be *true* or *false*. (To make sure we enter the if or else part of an if-else statement.)

For finding Constraints we also use an Antlr4 listener. This listener builds up a stack of lists of constraints. This structure is convenient for us, since it allows us to throw away requirements that are irrelevant to our sought after result easily by popping the stack. We push a new list onto the stack when we enter a scope, and we pop the stack any time we exit a scope or block for which it turned out that the sources and tests are irrelevant to our goal.

We take special care when exiting an if statement, because if there is an else statement after the if statement the guard for the if block needs to be inverted and saved as guard for the else block.

The saved constraints are found by the listener when entering "guarded" scopes like if, else, and while statements, as well as assert statements. Which can be seen as "guards" for the whole rest of the method.

In our example the first constraint is a "source" constraint for the parameter. The second and last constraint is the return statement. Because this return statement is the one that Viper pointed to as cause we stop searching for new constraints.

The stack structure laid out above is built as follows. First we enter the class, which is the first scope, then we enter the method, which is the second scope. In that second scope the parameter "source" constraint is placed, as well as the return "restriction" constraint.

3.5 Method Parameter Extraction

To find the required values of method parameters we again make our way to the Java Parser. Since we have access to the return statement that causes the error (We found this when interpreting the Viper Error in section 3.3) we can find the method that our error occurs in. We do this by recursively finding the parent of the parser rule, and checking if that is a method definition. A return statement in a well formed Java file is always inside a method definition[7], so this will eventually find the method definitions Parser Rule.

From this parser rule we can take the parser rule that defines the parameters. This is either empty (no parameters) or a structure of tuples that either contain a parameter and a reference to the next tuple, or just a parameter. (One could compare this to a linked list.) From this we extract a list of method parameters.

In our example we have one parameter. From the parser rule that defines the parameters we see a linked list node-like object with one parameter, and no reference to the next. From which we conclude that there is only one parameter, which we return.

3.6 Finding Involved Instance Variables

We find involved instance variables from another perspective. Instead of starting at the parsed file we start at our constraints, and extract the involved variable from those, for every variable we check to see if it has a member declaration. If so, this is an instance variable. By going from this direction we make sure that we only perform calculations on variables that are actually constrained.

Our example does not contain an instance variable, so for this section we will act like the method parameter "inp" is an instance variable, and the return statement is "return this.inp;".

Going through the constraints we see the constraint from the return statement that points to variable "inp". We look through the instance variable declarations for the "example_1" class and find that there is indeed a variable declaration named "inp". Since there are no other constraints we return a list of this one instance variable.

3.7 Resolving Required Values

Now that we have found all guards, instance variables, and method parameters we can move on to finding their required value. There are several cases possible here.

The first case is where the variable is the "goal". That is, it is the variable name that must eventually become *null*. The value for this variable is always *null*. The cases where this variable also has requirements are not supported yet.

The second case is where the variable is not the goal, and also has no constraints. In this case we make a default value for a its type. (See section 3.8 below.)

The third case is where the variable is not the goal, but it does have requirements. We then get the requirement, and make it create a value. Since for now the only supported constraints are simple *if (varname)* constraints this task is easy. We return the boolean *true* or *false* that we have stored as required value. Inverted if marked to be so.

Only being able to handle one requirement per variable is a significant limit, more about this in section 5.3 on page 6 about the limits of this tool.

In our example we see one variable that we need to find a value for, which is the method parameter. This is also the "goal". That is, it is the variable that should be null when it is returned. So, we set the source for this variable, the first method parameter, to null.

3.8 Default Values for Parameters Without Constraints

The default value for a type is a relatively simple affair. Most numeric types return 1 in their respective type. The default String is empty, and the default Character is a lowercase a. For the float and double we chose 0.0. These values have not been chosen for any important reason apart from needing to look reasonably default. A value of $0.66F$ for a floating point method parameter could be quite confusing for the user.

3.9 Variable Initialisation

Java calls the setting of a variables value *initialisation*[7]. In order to do this in text we take the following basic format: *<type> <name> = <initialiser>;* We fill in the type with the textual representation of the type that the required value has. (This means that we only support initialising *boxed* primitives, but that is not a problem as Java can convert those to *primive* on the fly.) For most types we can simply take the *String* representation of a variable as initialiser. For the *Float* we append an 'F', and for *Character* and *String* types we have to surround the textual representation with the ' and " characters respectfully.

At this point our tool does not support initialising custom classes, as this was considered out of scope. Other projects that we know of (Like Java Spring) have methods to do this (like *Autowired*), which require help from the programmer to perform their task. We speak more about this in section 6 about recommendations for the future.

For our example only the method parameter needs initialisation. We find its type to be int[] from its definition as method parameter. The value that we find using the previously mentioned system is null. For which the initialiser is "null". We add all that together to get the text "int[] param0 = null;".

3.10 Code to Call a Method

First we use the method described above to set up the required variables. Parameters are named *param0*, *param1* and so on. If the method is an instance method we assume that it has a default constructor (one that takes no arguments [7]) and use that in the following structure: "*<classname> instance = new <classname>();*".

We can then use these known parameter names to create the code that calls the method. We start with an empty string. If the method is a static method we append the classname, if not we append "*instance*". We append a dot, the name of the method, and then opening parentheses. For as many parameters as there are we append "*param<N>, *". Making sure not to add a comma after the last parameter. We then add closing parentheses and a semicolon to end the line.

For both the static and nonstatic case we assume that the method does not have a private access modifier.

The method in our example is not static. So we have to create an object. We get the class name from the class definition and, (correctly) assuming that it has a default constructor, we generate the initialisation like so: example_1 instance = new example_1();.

We then use this object to call the method. Since it has one parameter we add the name of that parameter between the braces; instance.testing(param0);.

4. RESULTS

Before starting we formulated three example problems that our tool should support. We will now show those three examples. After which we show some more examples that show more advanced features.

4.1 Failure Dependent on a Missing Precondition

For this example we run VerCors on a method that directly returns its parameter. The result is annotated to be *non-null*, but the parameter is not. In the output our tool calls the method with *null* as parameter to show the issue.

```
public class example_1 {
  //@ ensures \result != null;
  public int[] testing(int[] inp) {
    return inp;
  }
}
```

Our tool outputs the following piece of code that shows that, when the first parameter is *null*, the postcondition is violated.

```
class MainCounterexample {
  public static void main(String[] args) {
    int[] param0 = null;
    example_1 instance = new example_1();
    instance.testing(param0);
    // Above will be null. But there is a
    // postcondition that claims it is not.
  }
}
```

4.2 Failure Dependent on Global State

For this example we run VerCors on a method that returns *null* if an instance variable is true. The result is annotated to be *non-null*. In the output our tool sets the instance variable to *true* and calls the method to show the issue.

```
public class example_2 {
  public boolean fail;

  //@ requires Perm(this.fail, 1);
  //@ ensures \result != null;
  public int[] testing() {
    if (this.fail) {
      int[] a = null;
      return a;
    } else {
      int[] a = new int[] { 1, 2 };
      return a;
    }
  }
}
```

Our tool outputs the following piece of code to show the user a global state that would invalidate the postcondition.

```
class MainCounterexample {
  public static void main(String[] args) {
    example_2 instance = new example_2();
    instance.fail = true;
    instance.testing();
    // Above will be null. But there is a
    // postcondition that claims it is not.
  }
}
```

4.3 Failure Regardless of Parameters or Global State

For this example we run VerCors on a method that returns *null* regardless of any state. The result is annotated to be *non-null*. In the output our tool calls the method to show the issue.

```
public class example_3 {
  //@ ensures \result != null;
  public int[] testing() {
    int[] result = null;
    return result;
  }
}
```

Our tool outputs the following piece of code to show the programmer that there is a null error regardless of global state or parameters.

```
class MainCounterexample {
  public static void main(String[] args) {
    example_3 instance = new example_3();
    instance.testing();
    // Above will be null. But there is a
    // postcondition that claims it is not.
  }
}
```

4.4 Failure Dependent on both Parameter and Global State

This example shows that requirements for the instance and the parameter state can be combined in one counterexample. We run VerCors on a method that returns the parameter if an instance variable is true. The result is annotated to be *non-null*, the parameter is not. In the output our tool sets the instance variable to true, the parameter to *null* and calls the method to show the issue.

```
public class example_4 {
  public boolean fail;
  //@ requires Perm(this.fail, 1);
  //@ ensures \result != null;
  public int[] testing(int[] inp) {
    if (this.fail) {
      return inp;
    } else {
      int[] a = new int[] { 1, 2 };
      return a;
    }
  }
}
```

In the output our tool creates an instance, sets the instance variable *fail* to *true*, the parameter to *null* and calls the method, which will return *null*.

```
class MainCounterexample {
  public static void main(String[] args) {
    int[] param0 = null;
    example_4 instance = new example_4();
    instance.fail = true;
    instance.testing(param0);
    // Above will be null. But there is a
    // postcondition that claims it is not.
  }
}
```

4.5 Failure Dependent on The Else Clause

This example shows that we can reverse requirements when the goal is to enter the *else block* of an if statement. We run VerCors on a method that returns *null* if an instance variable is not *true*. The result is annotated to be *non-null*. In the output our tool sets the instance variable to *false* and calls the method to show the issue.

```
public class example_5 {
  public boolean succeed;
  //@ requires Perm(this.succeed, 1);
  //@ ensures \result != null;
  public int[] testing() {
    if (this.succeed) {
      int[] a = new int[] { 1, 2 };
      return a;
    } else {
      int[] a = null;
      return a;
    }
  }
}
```

In the output our tool creates an instance, sets the *succeed* variable to *false* and calls the method, which will return *null*.

```
class MainCounterexample {
  public static void main(String[] args) {
    example_5 instance = new example_5();
    instance.succeed = false;
    instance.testing();
    // Above will be null. But there is a
    // postcondition that claims it is not.
  }
}
```

5. LIMITATIONS

In this section we discuss the limitations that our implementations has. The next section continues this topic with a discussion about about recommendations for future development.

5.1 Only Java is Supported

The VerCors toolset has support for a whole host of languages. Java, C, OpenCL, OpenMP, and its own Prototypal Verification Language PVL. It supports these languages by translating and simplifying features from those languages into a common underlying structure. (Similar to an AST) Due to a misunderstanding about the VerCors code we supposed that this structure did not contain any *method bodies*. Which would mean that the analysis that the author needed to do would not be possible. Based on this faulty knowledge we chose to limit ourselves to Java code. When the tool was almost done, too late in the research process to change this inherent flaw, did we realise that this supposition was not true. And the decision was made not to restart in the last several weeks, but to continue working within this limit.

5.2 Understanding Expressions

To get to the faulty expression as found by VerCors our tool needs to be able to understand expressions. For this we mainly use two systems. One for getting a name, and another for getting a constraint for the variable with that name, from an expression. (Discussed in section 5.3.) This is an inherently limited approach for several reasons.

One of those limits is the fact that the current architecture only supports expressions with one name in it. This can be resolved, but it is a limiting factor in the current state of the tool. We describe two possible solutions for this in our recommendations for the future. (Sections 6.1 and 6.2.)

Another limit comes from one of the abstraction layers we built. We could split the system into (roughly) two stages. One where we gather the information we need, and the other to translate this information into names and required values. In this second stage, when we are at the point that we try to gleam knowledge from the gathered *constraints*, we need to look up information about the surroundings of these constraints. This knowledge was already available at the point that we created these abstracted objects, this abstraction is not an undeniable success. This is not so much a limit on the capabilities of the tool in an inherent sense, but it is a limitation regarding development speed and easy of future development. (Further described in sections 6.1 and 6.2.)

5.3 Supported Expressions

Our tool currently gathers any expression that limits the flow of the program on its way to the return statement. But the stage that interprets these expressions is not able to interpret them all.

At this point our tool only supports guards for if and assert statements in the form of simple expressions that consist of one variable name, instance or local, or parameter, that evaluates to a boolean value. These may be required to be *true* or *false*. For the value of a return statement we also only support expressions in the form of a variable name, though these can be of any type. And must, due to the goal of this tool, always evaluate to *null*.

Our recommendations about this limitation are further described in sections 6.1 and 6.2.

5.4 Constructors and Public Variables

For calling the method that we need to call we serve both the case where the method is in a static environment, and the case where the method needs to be called on an instance. For both cases we assume that the method does not have a private access modifier. For the *non-static* method case we make two more assumptions. First we assume that there is a public default constructor for the type involved. This means that an instance can be constructed without parameters[7]. We also assume that any members that need to have a value are public. Making a system that tries to find setters for certain fields if they are non-public is certainly possible. Either by using *Reflection* or with more static analysis. More about this in section 6.4 about future development.

6. RECOMMENDATIONS FOR FUTURE DEVELOPMENT

The tool that we have created functions as a prototype with some major areas ready for improvement that, for time or scope reasons, have not been expanded upon in this version. In the following sections we write about all these sections and will lay out our recommendations for the future.

6.1 Symbolic Execution

Our tool contains a rudimentary system for symbolic execution. This system could very well be expanded in the future with more capabilities, it is built with that in mind, but future authors may also look into other tools for symbolic execution, as there is already a whole host of well known and powerful systems that can do this[2].

6.2 Understanding Expressions

Hand in hand with finding the required values of variables goes finding the constraints for these variables. This project could be expanded with more support in that manner, as the skeleton is already in place. Future authors may also choose to use another system for this task, possibly one that already knows how this works. They may look in the direction of JetBrains' Intellij[4] or FindBugs[1] for projects that parse and "understand" Java.

6.3 Custom Type Initialisation

This project has no support for custom type initialisation, though the we do recognise this field as one where advancements could be helpful for the industry. As we noted before, one prominent project that performs custom type initialisation is Java Spring. Said project is not only relevant due to using Java, their size also makes them an interesting target for improvements in library capabilities. In 2020 the Stack Overflow Developer Survey[10] found that 16.4% of the more than 42 thousand respondents state that they use the Java Spring web framework.

Java Spring's solution to custom type initialisation is the *Autowire* and *Bean* system. The programmer registers a way to create a new instance of a type based on some inputs, and as long as those inputs also have a registered factory, or are types that Spring knows to make out-of-the-box, this type will be created during runtime.

Java Spring's use of this feature clearly shows a market for a tool that can generate the initialisation of custom types with even less input from the programmer. A feature like this would be an upgrade to our tool, to Java Spring, as well as any other projects that require automatic type initialisation.

6.4 Private Variables

As we described in section 5.4 about constructor and public variable assumptions, our tool assumes that any method and constructor involved is available and has a public access modifier. Especially the access modifier limit is a point for future development. One could use static analysis to find the real access modifier of variables. Other projects like Jackson from fasterXML (which (de)serialises JSON and Java objects) assume that the variable is either public or looks for setters[9]. But that may not be viable for this project, since we may not be in a position to put requirements on the style of input code.

6.5 Code Generation

Code generation for our project is currently an ad-hoc system that appends Strings together to gain a result. VerCors already includes a facility for more advanced AST-like to source requirements, with support for other languages as well. It would certainly be a valuable addition to our tool to be able to generate source with this system instead of our own.

7. CONCLUSION

This paper gives a concise overview of how our tool works, as well as a number of examples to show what it can do. We also explain what the limits of its capabilities are, and, using these limits, give recommendations for future development of our tool.

The code for this tool can be found in the Authors fork of VerCors. (https://github.com/Matthias-Sleurink/vercors) Especially in the src / main / java / vct / experiments / test_generation folder.

The examples, with their generated output, can be found in the same repository in the examples / test-generation folder.

8. ACKNOWLEDGMENTS

I would like to thank both of my supervisors Prof. Dr. M. Huisman and Dr. P. Van Den Bos for their advice during every phase of creation. Their input was especially valuable for getting the correct things on paper in the right way. My last thanks go out to my brother Benjamin for proofreading the final version of this paper.

9. REFERENCES

[1] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, 2008.

[2] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), 2018.

[3] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. The VerCors Tool Set: Verification of Parallel and Concurrent Software. In *IFM*, volume 10510 of *Lecture Notes in Computer Science*, pages 102–110. Springer, 2017.

[4] JetBrains. IntelliJ. https://www.jetbrains.com/idea/, 2007.

[5] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 41–62. Springer-Verlag, 2016.

[6] Neil Mitchel. Hlint. https://github.com/ndmitchell/hlint, 2006.

[7] Oracle. *Java Language Specification.* Oracle, se16 edition, Mar. 2021.

[8] T. Par. Antlr4, 2012.

[9] T. Saloranta. *Jackson-Databind*, 2019.

[10] Stack Overflow. Stack Overflow Developer Survey, 2020.

[11] The Clang Team. ClangFormat. https://clang.llvm.org/docs/ClangFormat.html.

[12] B. Wichmann, A. Canning, D. Marsh, D. Clutterbuck, L. Winsborrow, and N. Ward. Industrial perspective on static analysis. *Software Engineering Journal*, 10(2):69, 1995.