Improving Support for Java Exceptions and Inheritance in VerCors

Bob Rubbens, BSc.

Formal Methods and Tools Group
University of Twente

22nd May 2020

Abstract

In the age where one software bug can cost millions, software correctness is paramount. Static verifiers are used more and more in both academia and industry to prevent these costly bugs. They can formally prove that an implementation adheres to a specification. With the recent increased use of concurrency, proving correctness of software has become more challenging. However, progress is being made in this area: several static verifiers can now also verify languages in concurrent environments. Unfortunately their features are lagging behind: most checkers do not proceed beyond the prototyping phase and do not tackle the more practical language features. To improve the situation, this work presents an approach for implementing verification support for exceptions and inheritance as presented in Java. We also present, in great detail, the transformation of a language with exceptions and inheritance into a language without, and discuss the theory underlying the practical support for exceptions and inheritance. Finally, we briefly evaluate the approaches for both exceptions and inheritance, and discuss what can be further improved in static verification.

Contents

Chapter 1

Introduction

Java is a well-known technology in the software development industry. First released in 1996, it has since made its way to the upper half of the list of most used programming languages. According to the TIOBE index for March 2020, it is the most used programming language [71]. While indices like these should always be taken with a grain of salt, there is definitely reason to believe Java has a big impact on the lives of many software developers.

Multithreading in software has also become more and more important since the 90's. Multithreading combines concurrency and parallelism: it interleaves execution of multiple threads that are executed in parallel. While it has always had the important job of potentially improving software performance, it has become even more important in the age where Moore's law is not as strict as it once was [73]. Multithreading can be effective at speeding up certain kinds of workloads. However, it does come at a cost: increased software complexity. Where previously Java developers only had to worry about one specific execution of the program, multithreading exponentiates this problem. With multithreading it is possible that these problems no longer happen consistently between executions, but only happen in certain interleavings of certain situations. In the worst case, they might not even be encountered during development at all, and only appear in production.

A general trend can thus be observed: as more concurrency is introduced in an application, its complexity grows and debuggability plummets.

Taming complexity and ensuring correctness have always been goals that most programming languages pursue. However, programming languages also need to be practical, and hence compromises need to be made that can make these goals difficult to achieve. To bridge this gap between practicality and correctness, static verifiers have been developed. Static verifiers can reason about a program mathematically, and ensure that it adheres to a specification without running it. This allows for programs to be debugged before they are run, and increases the chance that bugs will be caught before software is deployed.

One of such verifiers is VerCors. VerCors is a static verifier for concurrent languages developed at the University of Twente [70]. It has frontends for Java, C, OpenCL and OpenMP, and focuses on verification of data-race freedom and functional correctness. Internally it uses separation logic to reason about multi-threaded accesses to data.

If we have these tools that can verify if a program is free of bugs, why do we still have bugs? In our opinion, part of the reason is that static verifiers are lacking support for language features that are often used in the industry. Two examples of such features, and the main focus of this work, are exceptions and inheritance.

1.1 Exceptions & inheritance

Exceptions and inheritance are two widely known features of Java. Most developers use them daily, sometimes without even realizing it.

Exceptions are used to identify and handle failures of many kinds in Java code. Osman et al. indicate that for four mature Java projects the proportion of exception-related code remains around 1%, even after 6 years of ongoing development [52]. This might not seem like a lot, but this means that for every 99 lines of code, there is 1 line of exception-related code! For code bases like Hadoop and Tomcat, which contain millions of lines of code [32, 33], these are significant numbers.

Exceptions allow the programmer to indicate in the program where an error might happen, and if it does, how it should be handled. Java has some support for checking at compile time if some exceptions are handled, but not for all of them. Exceptions are intended to make error handling more structured and software more robust, but they currently fail at the latter: 20% of reported bugs in 656 Java projects are related to improper exception usage [64].

Inheritance is used in Java for code reuse, and to indicate that some class specializes some other class. There are no data that indicate this feature causes bugs directly, but inheritance is integral for idiomatic Java. Dantas and Almeida Maia find that roughly 50% of classes in over 1000 GitHub and SourceForge projects either extend a class or implement an interface [16].

As a practical example, code that starts multiple threads of execution must either:

- Extend the `Thread` class.
- Implement the `Runnable` interface.

Without resorting to platform-specific APIs (which Java specifically wants to avoid), it is otherwise impossible to start another thread without inheritance [41,

p. 563]. Another idiomatic use of inheritance is to allow users to override some parts of a library and keep others, thus showing a clear path when functionality of a library needs to be extended.

It is clear that exceptions and inheritance are the cornerstones of idiomatic Java code. Exceptions are the main tool for error handling, and if developers want to work with threads it is impossible to get around inheritance. Therefore, if practical Java code needs to be verified, support for exceptions and inheritance is mandatory.

Supporting exceptions and inheritance in sequential environments is relatively well understood. However, it is still new ground for verifiers supporting multithreading. This is because it is not trivial to integrate inheritance and exceptions with verification techniques such as separation logic, which is typically used for reasoning about concurrent programs. Therefore, among other factors, these two features make application of static verification tools to commercial software difficult. Combined with the increased use of multithreading in the industry, it will only become more difficult to apply static verification to existing commercial software, unless these two major language features are soon supported by most static verifiers.

1.2 Objectives

To change the status quo, two things must happen.

First, support for static verification of multithreaded programs must be extended. It is no longer enough to verify only the sequential subset of a language, or a small concurrent subset. The industry needs support for a practical concurrent subset as soon as possible.

Second, the theory for these techniques needs to be researched and documented. The concrete implementation of the theory has to be properly documented as well. This is not only important for academics who want to advance the theory of verification, but also for developers who are going to use the tools.

This work aims to improve the situation for both objectives.

1.3 Research questions

This research aims to research and document support for inheritance and exceptions in VerCors. Therefore we decided on the following main research question:

How can VerCors support Java inheritance and exceptions?

We answer the main research question by dividing it up into three sub-questions:

SQ1: What are the state of the art techniques for static verification of exceptions and inheritance?

SQ2: In which ways is the state of the art incompatible with a concurrent environment, and what can be changed to amend this incompatibility?

SQ3: How can these techniques be implemented in VerCors, and if not, what is preventing this?

We will now elaborate on why these sub-questions have been selected.

First, it needs to be determined how other verifiers handle verification of exceptions and inheritance. This is important to reuse results about the theoretical foundations of exceptions and inheritance, as well as implementation guidelines.

Second, approaches to verify exceptions and inheritance must be designed for VerCors. To achieve this, existing implementations of support for exceptions and inheritance need to be analysed. Their trade-offs and limitations need to be compared. If possible they can provide a starting point for the implementation in VerCors.

Some implementations might be agnostic to whether or not the environment is concurrent or not. A proof or rigorous argument that this is the case would be desirable if the technique will be reused in VerCors

Third, it needs to be determined if the currently known approaches are compatible with both the verification approach of VerCors and its backend Viper. If this is not possible, then what is preventing it and how can this be resolved? For this sub-question, both theoretical foundations and practical limitations are relevant. This is because the theoretical foundations provide opportunities for future work, and the practical limitations are important to know for the end-users. Furthermore, any changes that can be made to the approach to improve decoupling between VerCors and Viper will be useful as well from an architectural point of view.

1.4 Approach

To answer the individual sub-questions, several tasks were performed.

To answer **SQ1** a literature review was done. This review includes an overview of static verifier functionality, as well as functionality regarding verification of exceptions and inheritance. Both old and new verifiers are considered. This is important, because it ensures older sequential approaches as well as newer concurrent approaches for exceptions and inheritance are included. While older sequential approaches are often not directly applicable to a concurrent environment, they might yield useful insights. These results can be found in Chapters 6 and 9.

To answer **SQ2** the approaches found in the literature review were analysed for their usefulness in concurrent environments. Sequential approaches were also taken into account by investigating why they were not compatible with a concurrent environment if so. These results can be found in Chapters 7 and 8.

To answer **SQ3**, the results of **SQ2** were evaluated in the context of the VerCors architecture. Then the approach that fits best in the VerCors architecture was designed with the approaches from **SQ2** as starting point. These results can also be found in Chapters 7 and 8.

1.5 Contributions

This work presents the following contributions:

- An overview of older and newer tools and a discussion of their support for concurrency and practical language features like inheritance and exceptions.
- A novel encoding of Java with exceptions into Java without exceptions compatible with a concurrent environment.
- A detailed practical description of support for inheritance based on the work of [53], [36], and [54].
- A prototype implementation of the presented approach for exceptions in VerCors.
- A manual evaluation of the presented approach for inheritance in VerCors.

1.6 Overview

Chapter 2 discusses unexpected details in the Java programming language, as well as behavioural subtyping for inheritance in Java. Chapter 3 discusses the principles of deductive verification, and the elements used for specification of Java code. Chapter 4 discusses knowledge needed to understand the theory and practice of verification of concurrent programs with separation logic. Chapter 5 discusses the tools that are the main focus of this work, showing how the tools are used in practice and discussing concepts specific to these tools. These tools are VerCors and Viper. Chapter 6 presents an overview of the state of the art in static verification. Chapter 7 presents the theory and implementation of verification support for exceptions in Java. Chapter 8 presents the theory and a manual evaluation of verification support for inheritance in Java. Chapter 9 compares the approaches outlined in this work to approaches implemented in several other verifiers, such as Verifast, Nagini, and jStar. Finally, we give our conclusion in Chapter 10.

Chapter 2

Java

The Java programming language is ubiquitous both in industry and academia. Therefore, in this work we assume the reader has basic familiarity with the syntax and semantics of Java, exceptions and inheritance. However, for completeness, a brief overview of exceptions and inheritance is given. For a more accessible introduction to Java, we refer the reader to the free online textbook *Introduction to Programming Using Java* by Eck [22].

We first discuss the notation used in this chapter and the rest of this work for Java snippets. Then we give an overview of exceptions and discuss the technical details. Finally, we give an overview of inheritance and discuss inheritance through the formal definition of behavioural subtyping.

2.1 Notation

At several occasions in this work Java code snippets are presented. Some of them are not complete Java programs. They are instead intended as chunks of a larger hypothetical Java program. The snippets are presented this way for an economical presentation. For example, a snippet might consist of a method definition followed by several statements, such as in Listing 2.1. This should not be interpreted as a faulty Java program, but a Java program that contains both the method, and the sequence of statements somewhere in the program.

Another notation that is used is that a method implementation is replaced by just a semicolon `;`. For example, in Listing 2.1 the implementation of `write` has been replaced with `;`. Although this looks like an abstract method, this is not the intention. Instead, it is intended as a concrete method for which we did not specify an implementation because the implementation is not the focus of the example.

Listing 2.1: Example of an incomplete method

```
int[] get_elems() {
  return new int[0];
}

void write(int[] buffer, int length);

int[] buffer = get_elems();
write(buffer, 10);
```

2.2 Exceptions

In this section, we give a brief example of general exception usage. Then we discuss how exceptional control flow can be grouped with break and return under abrupt termination. Finally, we discuss an edge case of `finally`.

2.2.1 Usage example

In Listing 2.2, data is written to a buffer. However, writing this data may fail. This is indicated by the `throws IOException` attribute on the `write` method of `Buffer`. If writing fails, the call to `write` on line 7 throws an instance of `IOException`. This exception is caught by the `catch` clause on line 8. The `catch` clause logs an error that writing failed. In either case, whether the call to `write` throws or not, the `buffer` is closed by calling the `close` method. This is enforced by the `finally` clause, which is always executed, even if an exception is thrown or the method returns.

Listing 2.2: Example usage of `try-catch-finally`

```
 1  class Buffer {
 2    void write(int[] data) throws IOException;
 3    void close();
 4  }
 5
 6  try {
 7    buffer.write(data);
 8  } catch (IOException e) {
 9    Logger.logError("Could not write data to buffer");
10  } finally {
11    buffer.close();
12  }
```

2.2.2 Abrupt termination

Abrupt termination [47, p. 14] is a grouping term for control flow that does not go from one statement to the next, like regular control flow. Instead, abrupt termination is when a statement terminates not because it is completed, but because it is terminated sooner than normal and control flow is redirected to another program point. Abrupt termination is sometimes also referred to as non-local or non-linear control flow.

One example of abrupt termination is the `throw` statement, as it aborts execution of the current block and redirects control flow to the nearest `catch` block. However, `break`, `continue` and `return` are examples of abrupt termination as well: they all terminate the current block earlier than normal, and redirect control flow to another program point.

We also want to mention labeled `break` and `continue`, as they are a source of complexity for this work. Labelled `break` and `continue` allow the user to specify which loop to `break` from or `continue` to. These constructs can be useful when nested loops are used. By allowing use of labels these features become similar to `goto`. Furthermore, labeled `break` can also be used to `break` from compound statements, such as `if`, `switch` or blocks.

2.2.3 Abrupt termination hiding

In Java, the `finally` clause is executed when control flow leaves the `try` block. However, since `finally` can contain arbitrary statements, it can overwrite reason the `try` block was terminating (for example, an exception, or `return` statement), thus hiding information.

Listing 2.3: Example usage abrupt termination hiding

```
1 int write() throws IOException {
2   try {
3     throw new IOException();
4   } finally {
5     return 3;
6   }
7 }
```

The example in Listing 2.3 illustrates this problem. The exception thrown on line 3 would normally propagate to the caller. However, because the `return` statement on line 5 is executed after the `throw`, it overwrites that an exception is being thrown, thus hiding the exception. While it is easy to avoid putting a `return` statement in a `finally` clause, this behaviour becomes problematic once methods are being called in `finally` that can also throw. In the worst case, an `OutOfMemoryError` might be unintentionally hidden, allowing the program to continue in a broken state.

This problem is partially resolved by the `try-with-resources` construct. This statement saves any exceptions that occur while closing the resource in a list contained in the original exception.

2.3 Inheritance

We now give a brief example usage of inheritance, and then show how inheritance can be formalised through behavioural subtyping.

2.3.1 Usage example

In Listing 2.4 the class `Cell` allows setting and getting the field `value`. A developer wants to extend the behaviour of `Cell` by keeping track of the previous `value`. This can be achieved by creating a new class, `ReCell`, that extends the `Cell` class using `extends`. In this specific case, `ReCell` specializes the behaviour by keeping track of the previous value in a separate field `bak`. In `set`, it still uses the `set` implementation of `Cell`. `ReCell` also does not override `get` as the implementation can be reused. By doing this, `ReCell` respects the behaviour of the superclass `Cell`: `set` still sets, and `get` still gets.

Listing 2.4: `ReCell` extends `Cell` through inheritance. This example was originally described by Parkinson in *Local reasoning for Java* [54]

```
class Cell {
  int value;

  void set(int x) {
    value = x;
  }

  void get() {
    return value;
  }
}

class ReCell extends Cell {
  int bak;

  void set(int x) {
    bak = value;
    super.set(x);
  }
}
```

2.3.2 Behavioural subtyping

Java sets no limitations on how inheritance is applied in practice. Therefore, it can be used for both code reuse and specialization. Use of inheritance falls in the category of code reuse if it is used to, for example, inherit the implementation of a method of another class. However, if the only goal is reuse, it is often not the case that the subclass respects the behaviour of the superclass. Inheritance falls in the category of specialisation if the subclass still respects the behaviour of the subclass.

This notion of "respecting the behaviour of the subclass" is formalized through behavioural subtyping. It is defined by the Liskov substitution principle [46], also referred to as the Subtype Requirement:

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Informally, this can be interpreted as stating that given x is type T, y is type S, and S is a subtype of T, then every x can also be replaced by y.

In this work, whenever it is stated that some type C is a behavioural subtype of D, it is implied that the Liskov substitution principle would hold for types C and D. For example, in Listing 2.4, it is the case that `ReCell` is a behavioural subtype of `Cell` because any place where `Cell` can be used, `ReCell` would also work.

Chapter 3

Deductive Verification

This chapter discusses deductive verification. We start by discussing the theory of deductive verification in logic, as well as Hoare triples. Then we apply this knowledge to verification of Java programs, and discuss the specification elements needed for verification of Java programs as well as the semantics of these specification elements.

3.1 Deductive verification

Deductive verification is the approach of using deductive logic for verification of programs. Therefore, the term deductive verification raises two questions: what is deductive logic, and how is it used for program verification. These questions will be discussed in sequence. For a more thorough discussion of these topics we refer the reader to *Mathematical logic for computer science* by Ben-Ari [6].

3.1.1 Deductive logic

Deductive logic is the process of reasoning from several facts towards a conclusion, using axioms and rules. The sequence of applied rules and axioms used can be seen as a proof for the conclusion.

We will now discuss an example of such a proof. We begin with the logical statement:

$$\textit{the sun shines} \wedge \textit{sky is blue}$$

Intuitively, to prove this we need to prove that both the sun shines and that the sky is blue. Formally, we can use the rule $\wedge$-split:

$$\frac{P \qquad Q}{P \wedge Q}\ \wedge\text{-split}$$

This rule states that to prove $P \wedge Q$, it suffices to prove both P and Q individually. The elements above the line are called premises, and the elements below the line the conclusion. A rule with no elements above the line can be applied anytime: it is an axiom. Applying the rule to the logical statement by replacing P and Q with the elements from the logical statement yields the following:

$$\frac{\textit{the sun shines} \qquad \textit{the sky is blue}}{\textit{the sun shines} \wedge \textit{sky is blue}}\ \wedge\text{-split}$$

Proving that the sun shines might involve complex logic, requiring a separate proof on its own. This is indicated by four vertical dots. However, the sky being blue is because of Rayleigh scattering, and therefore an axiom in the logical system. Applying this notation results in the following finished proof:

$$\frac{\begin{array}{c}\vdots\\ \textit{the sun shines}\end{array} \qquad \dfrac{}{\textit{the sky is blue}}\ \text{Rayleigh scattering}}{\textit{the sun shines} \wedge \textit{sky is blue}}\ \wedge\text{-split}$$

The deductive logic described in the example is limited to plain logic. Deductive logic can be extended to allow reasoning about programs with Hoare logic, as discussed in the next section.

3.1.2 Hoare logic

To extend deductive logic to allow reasoning about programs, deductive logic is extended with Hoare triples. This is also called a Hoare logic, and was first introduced by Tony Hoare in ‘An Axiomatic Basis for Computer Programming’ [30]. A Hoare triple has the following form:

$$\{\ P\ \}\ c\ \{\ Q\ \}$$

The Hoare triple states that if P holds, and c is executed, Q will hold. Conversely, if P does not hold and c is executed, anything can happen. P and Q are referred to as the pre-state and post-state respectively. c can be an atomic statement, or a compound statement consisting of other statements. Deductive rules can now be defined that prove or decompose Hoare triples, allowing to prove some programs correct. An example of a rule that decomposes a Hoare triple is the Sequence rule:

$$\frac{\{\ P\ \}\ c\ \{\ R\ \} \quad \{\ R\ \}\ d\ \{\ Q\ \}}{\{\ P\ \}\ c; d\ \{\ Q\ \}}\ \text{Sequence}$$

This rule states that to prove that Q holds after c and d are executed in sequence, c and d need to be proven to hold individually. R is the expression that must hold in-between c and d, and can be chosen appropriately.

An example of an axiomatic rule is the rule for assignment:

$$\frac{}{\{\ Q[E/x]\ \}\ x := E\ \{\ Q\ \}}\ \text{Assign}$$

The notation $Q[E/x]$ means "Q where every occurrence of x is replaced by E". Thus, this rule states that if Q with every occurrence of x replaced by E holds, then Q holds after the assignment of E to x. For example, we can apply this rule to the statement $x := 4$ to get the following proof:

$$\frac{}{\{\ (x = 4)[4/x]\ \}\ x := 4\ \{\ x = 4\ \}}\ \text{Assign}$$

Since $(x = 4)[4/x] = (4 = 4) = \mathit{true}$, the pre-state is always true. Hence, this assignment can always be proven correct, with respect to the pre- and post-state.

3.2 Java program verification

Java verification is often done in a notation introduced by JML. VerCors follows this notation, and also uses several specification elements introduced by JML. The notation requires that any specification statements are put directly in program code. However, directives are either preceded by `//@`, or surrounded by `/*@ ... @*/`. This effectively puts all verification directives in comments, ensuring the verification code has no runtime cost. Semantically, verification directives can only inspect the program state, but not change it.

Verification of Java programs is mostly done through the following specification elements: `assert`, `requires`, `ensures` and `loop_invariant`. VerCors allows use of other specification elements that do not exist in JML, such as `context` and ghost parameters. However, this work does not use them. We refer the reader to the VerCors documentation [72] for more info about these specification elements. In this section we discuss each of these verification elements, using the example shown in Listing 3.1. We also give an informal and incomplete description of what Hoare logic rules and triples would look like for these verification elements. The rules are incomplete in the sense that certain aspects of programming language semantics

Listing 3.1: Example usage of `assert`, `requires`, `ensures`, and `loop_invariant`.

```
1  void test() {
2    int input_v = 20;
3    //@ assert input_v > 0;
4    int output_v = incrementBy10(input_v);
5    //@ assert output_v == 30;
6  }
7
8  //@ requires v > 10;
9  //@ ensures \result == v + 10;
10 int incrementBy10_2(int v) {
11   int total_v = v;
12   int i = 0;
13   //@ loop_invariant total_v == v + i;
14   //@ loop_invariant 0 <= i && i <= 10;
15   while (i < 10) {
16     total_v = total_v + 1;
17     i = i + 1;
18   }
19   //@ assert !(i < 10);
20   return total_v;
21 }
```

are missing. For example, modelling of local and global state, expressions with side-effects and proper modelling of return values are missing. However, because they are incomplete does not mean they are not useful, as the examples give an intuition of what is verified. For a formal discussion of Hoare logic rules and triples for Java, we refer the reader to 'Java Program Verification via a Hoare Logic with Abrupt Termination' by Huisman and Jacobs [35].

3.2.1 Assert

The `assert` specification statement in VerCors is similar to the `assert` statement in Java, in the sense that they both fail if the condition is not true. However, the specification `assert` cannot have side-effects. For example, an `assert` that uses the increment operator (e.g. `i++`) is not allowed. Furthermore, it is statically checked by VerCors, and does not throw an exception at runtime. In this work, if the `assert` statement is mentioned without any qualification, the specification statement is intended, and not the Java statement. In Listing 3.1 `assert` is used on line 3 to check that the value of `input_v` is positive.

The Hoare rule for `assert` is as follows:

$$\frac{}{\{\ P\ \}\ \texttt{assert}\ P;\ \{\ P\ \}}\ \text{Assert}$$

Informally, it requires the condition P to hold before the `assert`, and allows P to be assumed afterwards.

3.2.2 Requires and ensures

`requires` and `ensures` denote pre- and post-conditions of a method. The `requires` and `ensures` elements are sometimes also referred to as the contract in general. `requires` clauses specify what needs to hold before the method can be called. Since they hold when the method is called, `requires` clauses can therefore be assumed at the start of the method.

Conversely, `ensures` specifies what needs to hold when the method finishes, either via `return` or `throw`, or by simply executing the last statement. Similarly, since `ensures` clauses hold at the end of a method, the `ensures` clause can also be assumed after a method call returns.

`requires` and `ensures` mirror the structure of a Hoare triple: they state what is required to hold before the method, and what is ensured to hold after the method is executed. The method implementation mirrors the proof of how post-condition follows from the pre-condition. This makes reasoning about methods & method calls similar to reasoning about Hoare triples.

The contract can also be seen as an interface: it abstracts away from the implementation. Even though an implementation might use a complex algorithm or GPU resources to compute the result, the contract ensures that clients can only depend on abstract properties of the result.

In Listing 3.1, the contract of `incrementBy10` is specified on line 8. The method requires that the argument `v` is bigger than 10, and in return it will ensure that the return value is the sum of the argument and 10. Note that the return value is referred to through the `\result` keyword. The postcondition of `incrementBy10` allows the assertion on line 5 to verify. This is because `input_v` equals 20, and the postcondition states the result is `v` incremented by 10. Therefore, `output_v` must equal 30.

The `requires` and `ensures` clauses appear in two Hoare rules: the Call rule and the MethodDefinition rule. They are defined as follows:

$$\frac{}{\{\ f.\texttt{requires}\ \}\ x.f(\bar{e})\ \{\ f.\texttt{ensures}\ \}}\ \text{Call}$$

$$\frac{\{\ P\ \}\ c\ \{\ Q\ \}}{U\ f(\overline{T\ x})\ \texttt{requires}\ P;\ \texttt{ensures}\ Q; \{\ c\ \}}\ \text{MethodDefinition}$$

Call inserts the pre- and post-conditions into the proof to be proven and assumed respectively. MethodDefinition requires a proof that given P, after c is executed, Q holds. Here U is the return type of the method, and $\overline{T\ x}$ represents a series of arguments.

Modular verification

The way `requires` and `ensures` are defined allows modular verification, which is an important property of static verifiers. Modular verification means that methods can be verified independently of the correctness of other methods [3]. During verification of a method, the contracts of other methods are assumed to be correct. The correctness of a method can then be verified. Whether or not those called methods actually adhere to their contracts can be verified independently at a later time.

Modular verification has three beneficial properties. First, method implementations cannot break other methods. If a method changes its implementation, other methods remain correct, as long as the contract remains unchanged. Second, verification of many methods can easily be parallelized, or earlier results cached. Third, an implementation of a called method can be omitted, since only the contract is needed for verification of other methods. This is useful in situations where there are multiple parties working on one piece of software, or if parts of the software are not implemented yet.

3.2.3 Loop invariants

A `loop_invariant` is used for the verification of loops. Loops can be verified in two ways. One way is to unroll the loop as many times as needed for the method to verify. This is a simple and effective method. The drawback is that it is often impossible to know beforehand how many times the loop needs to be unrolled. Therefore, unrolling the loop is an incomplete method for verification.

A different way is to use a loop invariant. The loop invariant is specified by the user, and is supposed to hold upon entry of the loop. The loop invariant also needs to hold after every iteration of the loop. After the loop terminates, the loop invariant can be assumed, together with the negation of the loop condition. By simplifying verification of the loop into checking if a loop invariant holds, verification becomes complete again: given a loop invariant, VerCors can check whether it holds upon entry and after an iteration. Conversely, loop unrolling is not complete, as it is not possible to tell if unrolling a loop one more time will allow the program to verify. However, the loop invariant has a cost: they can require some original insight from the user, making the verification process less automated.

Listing 3.2: General form of a `signals` clause.

```java
class E extends Exception {
  public int x;
}

//@ signals (E e) e.x > 0;
public int m() throws E { ... }
```

In Listing 3.1, the loop invariant on line 13 facilitates proving that `total_v` is indeed incremented by 10. The loop invariant is combined from 2 smaller loop invariants, which state that 1) `total_v` is always equal to the sum of `v` and `i`, and 2) that `i` stays between 0 and 10 inclusive. 1) holds after every loop iteration, as both `total_v` and `i` are incremented. 2) holds as well, since as soon as `i` becomes 10, we exit the loop. After the loop, the negation of the loop condition holds, as shown by the `assert` on line 19. This negation, combined with 2), implies that `i` must be 10. This fact, combined with 1), allows VerCors to prove the postcondition.

This is the Hoare rule for While:

$$\frac{\{\ C \wedge I\ \}\ c\ \{\ I\ \}}{\{\ I\ \}\ \texttt{while}\ (C)\ \texttt{loop\_invariant}\ I;\ \{\ c\ \}\ \{\ \neg C \wedge I\ \}}\ \text{While}$$

Note that c is verified in isolation: it cannot depend on state from before the loop. If information from the pre-state of the loop is required, it must be put in the loop invariant.

3.3 Exception specification

The JML `signals` clause is used in VerCors for the specification of exceptions. It indicates a postcondition that must hold when an exception of a specific type is thrown. It is also referred to as an “exceptional postcondition”.

The specific syntax for this clause is shown in Listing 3.2. The type `E` has to be a subclass of `Throwable`, as is the case in Listing 3.2. `e.x > 0` is the postcondition that must hold when the exception is thrown, and can state properties about the thrown object `e`.

For any type that a method throws, the method must have a `throws` attribute or a `signals` clause. Therefore, the set of types a method can throw is the union of types in its `signals` clauses and its `throws` attribute. This is a good property for verification as it is now possible to exactly know which types can be thrown at some point in the program.

However, this does not accurately model Java exception semantics, because unchecked exceptions can always be thrown. This uncertainty can be opted into by adding a `signals` clause with the `true` post-condition. For example, adding `signals (RuntimeException e) true;` ensures that any calling code must assume that *any* `RuntimeException` can be thrown. Therefore, any calling code that is verified must wrap this method call in a `try-catch` statement to ensure it does not escape, or indicate it throws a `RuntimeException` as well.

The Hoare rules for exception specifications modify the rules for Call and MethodDefinition. They introduce an additional variable *exc*, which is `null` when no exception is thrown, and non-`null` when an exception is thrown. The rules are:

$$\frac{}{\{\ f.\texttt{requires}\ \}\ x.f(\bar{e})\ \{\ excQ\ \}}\ \text{CallExc}$$

$$\frac{\{\ P\ \}\ c\ \{\ exc = \texttt{null} \implies Q \land exc \neq \texttt{null} \implies R\ \}}{U\ f(\overline{T\ x})\ \texttt{requires}\ P;\ \texttt{ensures}\ Q;\ \texttt{signals}\ R;\ \{\ c\ \}}\ \text{MethodDefinitionExc}$$

Where

$$excQ \equiv (exc = \texttt{null} \implies f.\texttt{ensures}) \land (exc \neq \texttt{null} \implies f.\texttt{signals})$$

3.4 Inheritance specification

For verification of inheritance no special syntax is needed. However, it still needs to be verified that the Liskov substitution principle, as discussed in Section 2.3.2, is adhered to. There are two options, each interpreting the substitution principle slightly differently and involving a different trade-off.

The first way is to verify that a method implementation obeys not just its own contract, but also each of its parent contracts individually. The advantage of this approach is that it is flexible, as it allows for code to be reused in multiple contexts. Consider Listing 3.3. The implementation of `EvenCounter` ensures `x` is even and remains so. The implementation can be reused for `OddCounter`, as long as `x` is initialized to an odd number. The drawback is that this approach is not modular: every subclass has to be verified in the context of every superclass, resulting in quadratic growth of proof obligations.

Listing 3.3: Example of a case of inheritance where flexible semantics are needed.

```
class EvenCounter {
  int x;

  //@ requires x % 2 == 0
  //@ ensures x == \old(x) + 2
  void count() {
    x = x + 2;
  }
}

class OddCounter extends EvenCounter {
  // New contract, same implementation:
  //@ requires x % 2 == 1
  //@ ensures x == \old(x) + 2
  void count();
}
```

The second way, often referred to as "specification inheritance", requires every method to adhere to the specification of the overridden method. This approach restores modularity, as for every subclass it only needs to be proven if it adheres to the contract of its superclass, resulting in a linear growth of proof obligations. It also transitively preserves the Liskov substitution principle in the inheritance hierarchy. The drawback of this approach is that it is less flexible: patterns such as Listing 3.3 are not expressible in this approach. However, because modularity is a valuable property for timely verification, specification inheritance is preferred over the non-modular approach. The verifiers OpenJML [14] and KeY [1] also use specification inheritance.

This is the proof rule for inheritance:

$$\dfrac{\begin{array}{c}\vdots\ \text{MethodDefinition}\\ V\ D.f(\overline{T\ x})\ \texttt{requires}\ R;\ \texttt{ensures}\ S;\end{array}\qquad \begin{array}{c} C\ \texttt{extends}\ D \qquad U\ \texttt{extends}\ V\\ \{\ P\ \}\ c\ \{\ Q\ \}\\ R \implies P \wedge Q \implies S\end{array}}{U\ C.f(\overline{T\ x})\ \texttt{requires}\ P;\ \texttt{ensures}\ Q;\ \{\ c\ \}}\ \text{MethodOverride}$$

It introduces several requirements when overriding a method. When overriding a method, the overridden method must exist. The method f in class D is indicated by the premise $D.f$. If $C.f$ is overriding $D.f$, it must also be the case that C is a subclass of D. The return type of the overriding method must be a subclass of the return type of the overridden method. The implementation of $C.f$ must adhere to the contract of $C.f$. And finally, the contract of $C.f$ and $D.f$ must be compatible: R must imply P, and Q must imply S. This allows $C.f$ to be used wherever $D.f$ is used, hence enforcing the Liskov Substitution Principle.

Chapter 4

Separation Logic

Separation logic is an extension of Hoare logic (discussed in Section 3.1.2). Its main purpose was to reason about sequential programs using pointers [50]. However, it was later determined that it could also be useful for concurrent programs, and was extended to concurrent separation logic by O'Hearn [51]. This chapter gives an informal introduction. For a more in-depth discussion of separation logic, we refer the reader to 'Separation logic' by O'Hearn [49], or the original papers referenced earlier.

Separation logic (SL) is discussed as it is the feature that allows verification of concurrent programs. Furthermore, it is particularly relevant for the verification of inheritance, as it prevents use of specification inheritance (as presented in Section 3.4). This problem is discussed in Section 4.2.4.

In this section, first the theory of SL is discussed: resources, permissions, the separating conjunction, and magic wands. Abstract predicates (APs) and abstract predicate families (APFs) are also discussed. Specific concurrency primitives from concurrent separation logic such as resource invariants and parallel composition are not discussed, as they are not relevant for this work. Then we discuss how these concepts can be used for the verification of Java, and what kinds of problems SL, APs and APFs solve.

4.1 Theory of separation logic

In this section we present a theoretical basis of SL. First, the basic elements of SL are discussed. These are the heap, which models which locations are accessible, and resource assertions, which make assertions about these locations. Then we move on to more advanced concepts: abstract predicates and abstract predicate families.

4.1.1 The heap

The heap is the context of a separation logic expression. In pure separation logic it is a partial mapping from addresses to values. This can be compared to an integer memory with linear address space. However, in the context of Java, the heap is usually considered as a collection of object fields with corresponding values, such as integers or references. This is also the view taken in this work. Besides a location and a value, an entry in the heap also contains a fraction between 0 and 1. If this fraction is 1 then the value can be written to. If the value is more than 0 and less than 1, it can only be read from.

Heaps can be split and merged. When split into two, fractions of entries must be properly divided between the two parts. When merged, the fractions of entries in both parts must be added together.

Formally, a heap is a partial function from locations to tuples of fractions and values:

$$h : loc \rightharpoonup (frac, val)$$

Concrete heaps can be expressed with the following notation. For a concrete heap h:

$$h \equiv \{\, l \overset{\frac{1}{2}}{\mapsto} v, \cdots \,\}$$

Here l represents a location or address, and v the value at that location. In this specific case, there is only read permission, as the permission is $\frac{1}{2}$. v can be replaced with an underscore (_) if the value is irrelevant.

4.1.2 Resource assertions

Resource assertions are expressions that make assertions about heaps. It can be checked if a resource assertion a holds for a specific heap h by evaluating the separation logic judgement: $h \models a$. If true, a holds for h. Resource assertions can also abbreviated to resources. We will now discuss what kind of resource assertions can be made about heaps.

Permission

Permissions $Perm(location, fraction)$ assert that an amount of permission is present in the heap for a specific location. $h \models Perm(l, f)$ is true if location l is in h with a fraction of at least f. Similar to fractions in the heap, $Perm$ can be split in two or merged. Some examples of permissions:

- $Perm(this.x, \frac{1}{2})$

- $Perm(obj.y, \frac{1}{1})$
- $Perm(z, \frac{2}{3})$
- $Perm(x.y.z, \frac{0}{1})$

Separating conjunction

The separating conjunction $P * Q$ composes resource assertions. $h \models P * Q$ holds if the heap can be split in two disjoint parts h' and h'' such that $h' \models P$ and $h'' \models Q$. Intuitively, this means that the permissions present in P and Q must not add up to more than 1: their "footprints" must not overlap . For example, the following judgement holds, as $\frac{1}{2}$ can be split into two quarters:

$$\{\ x \overset{\frac{1}{2}}{\mapsto} \_\ \} \models Perm(x, \tfrac{1}{4}) * Perm(x, \tfrac{1}{4})$$

However, the following does not, because not enough permission is present in the heap:

$$\{\ x \overset{\frac{1}{2}}{\mapsto} \_\ \} \models Perm(x, \tfrac{1}{2}) * Perm(x, \tfrac{1}{2})$$

Besides resource assertions, the separating conjunction also allows combining resource assertions and boolean expressions. For example, the following resource assertion states that the location x must have a value bigger than or equal to 0:

$$\{\ x \overset{\frac{1}{2}}{\mapsto} 5\ \} \models Perm(x, \tfrac{1}{2}) * x > 0$$

Since the heap contains the value 5 at location x, this judgement is true. The previous example exposes another important property of resource assertions: they must be self-framing. Self-framing implies that if a heap location is used in a boolean expression, the heap must at least have read permission for that location. This can be guaranteed by including a *Perm* in the resource assertion. The previous judgement is self-framing, but the next one is not, as y is not present in the heap at all:

$$\{\ x \overset{\frac{1}{2}}{\mapsto} 5\ \} \not\models y > 0$$

Furthermore, even if a resource assertion is properly self-framed, the judgement is still false if the heap does not contain the permission:

$$\{\ x \overset{\frac{1}{2}}{\mapsto} 5\ \} \not\models Perm(y, \tfrac{1}{2}) * y > 0$$

Separating implication

The separating implication, sometimes called magic wand, $P \mathrel{-\!\!*} Q$ indicates that a certain assertion can be gained by giving up another resource. This mutates the heap, as permissions are taken away and granted. Therefore, the separating implication is inherently imperative. We can express this exchange of resources in a Hoare rule for the *apply* statement:

$$\frac{}{\{\ P * P \mathrel{-\!\!*} Q\ \}\ \mathit{apply}\ P \mathrel{-\!\!*} Q\ \{\ Q\ \}}\ \text{Apply}$$

Doing this exchange of resources is often referred to as applying the magic wand. Note that after applying the wand, the magic wand itself is not in the post-state: applying the wand consumes it.

4.1.3 Abstract predicates

Abstract predicates (APs) were first introduced by Parkinson in *Local reasoning for Java* [54]. Their intended use is to enable encapsulation in separation logic. An abstract predicate definition consists of a name, zero or more parameters with types, and a body: $\mathit{pred}(\overline{T\ x}) = \mathit{body}$. An AP can be used in resource assertions, but is treated as an opaque resource. This means the resources contained in the predicate can only be used by explicitly exchanging the AP for the AP body. This is called “unfolding” the AP. After this exchange, the AP body is available, but the AP is not. In this sense APs are similar to magic wands, as they mutate the heap and hence are inherently imperative. This unfolding can be expressed in a Hoare rule for the *unfold* statement:

$$\frac{\mathit{pred}(\bar{x}) = \mathit{body}}{\{\ \mathit{pred}(\bar{e})\ \}\ \mathit{unfold}\,\mathit{pred}(\bar{e})\ \{\ \mathit{body}[\bar{e}/\bar{x}]\ \}}\ \text{Unfold}$$

Conversely, abstract predicate bodies can also be wrapped back into a predicate by folding. This is expressed in a Hoare rule for the *fold* statement:

$$\frac{\mathit{pred}(\bar{x}) = \mathit{body}}{\{\ \mathit{body}[\bar{e}/\bar{x}]\ \}\ \mathit{fold}\,\mathit{pred}(\bar{e})\ \{\ \mathit{pred}(\bar{e})\ \}}\ \text{Fold}$$

Like permissions, predicates can also be split and merged. The notation for this is to prefix an AP with a fraction in brackets:

$$\mathit{pred}(\bar{e}) \equiv [\tfrac{1}{4}]\mathit{pred}(\bar{e}) * [\tfrac{3}{4}]\mathit{pred}(\bar{e})$$

When a split predicate is unfolded, all permissions and predicates in the body are multiplied by the fraction of the predicate. Similarly, to fold a split predicate, only a fraction of the body is needed.

4.1.4 Abstract predicate families

Abstract predicate families (APFs) extend APs. Where abstract predicates are needed to allow encapsulation, abstract predicate families allow subtyping in separation logic.

Their notation is almost identical: definitions of APFs also consist of a name, argument and body. However, one change is that APFs are defined for one or more types. These types can be receivers of the APF. For example, if a type X has a predicate *pred*, it could be used in a resource assertion as follows, where x has type X:

$$\text{Defined for X: } \mathit{pred}(\mathit{int}\ a, \mathit{int}\ b) = \mathit{body}$$
$$h \models Perm(x, \tfrac{1}{2}) * x.\mathit{pred}(2, 3)$$

Defining an APF named *pred* for a type T introduces the following two elements for use in resource assertions. First, it allows the use of predicate family instances such as $t.\mathit{pred}()$. Second, it allows the use of the predicate family instance qualified with the concrete type T. Such qualified predicate family instances are called predicate entries, and their notation is: $t.\mathit{pred}@T()$.

APFs allow three interesting operations.

First, if the type of t is known, a predicate family instance may be exchanged with a predicate family entry of that type. Conversely, if a predicate entry is qualified by a type T and the receiver also has type T, it may be exchanged for a predicate family instance. Second, predicate entries can be unfolded and folded like regular predicates. Note that for folding and unfolding predicate entries, the type does not have to match explicitly. Third, the arities of predicate family instances can be adjusted as needed. If the last argument is not needed, it can be discarded. If an argument is missing at the end it can be existentially quantified and appended.

If multiple types define an APF with the same name, and the types are subtypes, the predicate family instances are shared. However, the predicate entries are still separate. For example, given objects t and u with types T and U respectively and T being a subtype of U, if they each define an APF $\mathit{pred}()$, then $t.\mathit{pred}()$ and $u.\mathit{pred}()$ can both be used. However, $t.\mathit{pred}()$ can only be exchanged for $t.\mathit{pred}@T()$, and not for $t.\mathit{pred}@U()$.

4.2 Usage in Java verification

Most of separation logic elements that are discussed in this chapter are already supported in VerCors. Therefore we discuss how separation logic is practically used up to and including APs. APFs are not yet supported in VerCors, but we discuss how they allow verification of inheritance. We also present a hypothetical usage example of APFs.

4.2.1 Syntax

The syntax for separation logic in Java is similar to the syntax used in the theory section. This is also the syntax used by VerCors. However, there are a few differences.

To make the distinction between SL in Java and pure SL clear, all Java SL elements are written in `monospace font` instead of *italic font* for pure SL.

Fractions for `Perm`s are written using a backslash, instead of a horizontal divider or the more conventional forward slash. For example, `Perm(this.x, 1\2)` and `Perm(y, 2\3)` are valid permissions, as they both use a fraction written with a backslash. `Perm(this.x, 0.5)` and `Perm(y, 4/5)` are not valid permissions, as decimal numbers and plain divisions are not fractions. The write permission of `1\1` can be abbreviated with `write`.

For locations, if merely a variable `x` is used, it is interpreted as if `this.x` was typed. This is similar to how general Java scoping works. For example, if the current class has a field `x`, `Perm(x, 1\2)` is identical to `Perm(this.x, 1\2)`. This also extends to abstract predicates: if a predicate `pred()` is used in a resource assertion, it is interpreted as `this.pred()`.

The separating conjunction is written with a double star, e.g. `P ** Q`, to avoid confusion with multiplication, `*`. The magic wand uses a dash and a star: `P -* Q`.

4.2.2 Permission usage in programs

In Java programs verified with SL, the heap is not explicit like the judgements from Section 4.1. Instead, the heap is implicit in the state of the program. This state is mutated when variable assignments are executed, objects are allocated through `new`, predicates are unfolded and magic wands are applied. Particularly, `Perm`s for locations can be acquired in the following ways:

- If available in the pre-condition, resources will be available at the start of the function.
- When objects are allocated, permissions for all the fields are created.

- By unfolding predicates and applying magic wands.

Permissions can also be leaked. For example, if a method does not put permissions in the post-condition, the permissions are thrown away at the end of the method and not returned to the caller.

Listing 4.1 contains some example usages of permissions in contracts and `assert`s.

Listing 4.1: Example usage of permissions.

```
//@ requires Perm(obj.x, 1\2);
//@ ensures Perm(obj.x, 1\2);
void returnsPermission(MyClass obj) {
  print(obj.x) // Read the value by printing it
}

//@ requires Perm(obj.x, write)
void leaksPermission(MyClass obj) {
  obj.x = 5; // Change the value
}

void setTwice() {
  MyClass object = new MyClass();
  //@ assert Perm(object.x, write);
  returnsPermission(object);
  //@ assert Perm(object.x, write); // Permission is still available
  leaksPermission(object);
  //@ assert Perm(object.x, write); // Permission was leaked: fails!
}
```

4.2.3 Java & abstract predicates

In Java, and software engineering in general, private fields of a class are not part of the interface. Therefore, clients of a class should not have to deal with private fields. However, in separation logic, clients do have to deal with the existence of private fields because the permissions to those fields have to be managed. An example of this is shown in Listing 4.2. Even though `value` is a private field, the permission to it still has to be managed by the client. Worse, if the implementation or permissions of `Cell` change, verification of the client might also break. Abstract predicates solve this by allowing permissions to be abstracted behind an opaque name.

Listing 4.2: Example of encapsulation being violated

```
class Cell {
  private int value;
  //@ requires Perm(value, write);
  //@ ensures Perm(value, write);
  void set(int x) {
    value = x;
  }
}

//@ requires Perm(c.value, write);
//@ ensures Perm(c.value, write) ** c.value == 5;
void setTo5(Cell c) {
  c.set(5);
}
```

In VerCors, abstract predicates are defined by specifying a ghost function with return type `resource`. In example Listing 4.3, this is done on Line 4. The `final` keyword is needed to indicate it is not an abstract predicate family, as discussed in Section 4.2.4.

Listing 4.3: Example of encapsulation in using abstract predicates

```
1  class Cell {
2    private int value;
3
4    /*@ final resource state(int p) = Perm(value, write)
5            ** value == p; @*/
6
7    //@ requires state(oldValue);
8    //@ ensures state(x);
9    void set(int x) {
10     //@ unfold state(oldValue);
11     //@ assert Perm(value, write);
12     value = x;
13     //@ fold state(x);
14   }
15 }
16
17 //@ requires c.state(oldValue);
18 //@ ensures c.state(5);
19 void setTo5(Cell c) {
20   c.set(5);
21   //@ assert [1\2]c.state(5) ** [1\2]c.state(5);
22 }
```

When only the name of the abstract predicate is used, as done on line 7 of the example, it is referred to as an abstract predicate instance. The right hand side of the definition is called the abstract predicate body, which is parametrised by the abstract predicate parameters.

Abstract predicates can be manipulated by the `fold` and `unfold` statements.

4.2.4 Java & abstract predicate families

Separation logic with abstract predicates allows encapsulation, but not subtyping. In Listing 4.4 `ReCell` is intuitively a subtype of `Cell`. However, their contracts are not subtypes, as a heap with one permission can never be the same size as a heap of two permissions. Abstract predicates would not resolve this problem: then there would be two incompatible abstract predicates instead of two differently-sized heaps. Therefore, in pure separation logic it is not possible to express subtyping.

Listing 4.4: Intuitively `ReCell` is a subtype of `Cell`, but the contract of `ReCell` is not compatible with the contract of `Cell`.

```
class Cell {
  int value;

  //@ requires Perm(value, write);
  //@ ensures Perm(value, write);
  void set(int x) {
    value = x;
  }
}

class ReCell extends Cell {
  int bak;

  //@ requires Perm(value, write) ** Perm(bak, write);
  //@ ensures Perm(value, write) ** Perm(bak, write);
  void set(int x) {
    bak = value;
    value = x;
  }
}
```

APFs enable verification of behavioural subtyping. In Listing 4.5, APFs have been used to model the common state between `Cell` and `ReCell`. In this example, if we check the contracts of `set` for both `Cell` and `ReCell` for subtyping, it is clear that they are subtypes. They both require the APF `state` and ensure the APF `state`. Hence, `ReCell` is a subtype of `Cell`. The management of predicate arities is done implicitly in this case. However, in a concrete implementation, it is likely these arities have to be managed manually, as APFs are currently not supported natively but are encoded into APs.

Listing 4.5: With abstract predicate families, it can be verified that `ReCell` is a subtype of `Cell`.

```java
class Cell {
  int value;

  //@ resource state(int p) = Perm(value, write) ** value == p;

  //@ requires state(oldVal);
  //@ ensures state(x);
  void set(int x) {
    //@ assert this instanceof Cell
    //@ unfold state@Cell(oldVal);
    value = x;
    //@ fold state@Cell(x);
  }
}

class ReCell extends Cell {
  int bak;

  /*@ resource state(int p, int q) =
        Perm(value, write) ** value == p
        ** Perm(bak, write) ** bak == q;
    @*/

  //@ requires state(oldVal, oldBak);
  //@ ensures state(x, oldVal);
  void set(int x) {
    //@ assert this instanceof ReCell
    //@ unfold state@ReCell(oldVal, oldbak);
    bak = value;
    value = x;
    //@ fold state@Cell(x, oldVal);
  }
}
```

Note that APFs as described in this chapter are not enough to support inheritance. In this description there is an implicit assumption that methods are only called if the dynamic type is the same as the class the method is defined in. However, this does not hold in the case of `super` calls and constructors. Approaches to deal with this are discussed in Chapter 8, as each approach involves different trade-offs.

Chapter 5

VerCors & Viper

The first section of this chapter discusses the interface VerCors offers for verification, its internals, and what an average session of VerCors usage looks like. The second section discusses the main backend of VerCors, Viper, and its input verification language, Silver.

5.1 VerCors

VerCors is a static verifier developed by the FMT group at the University of Twente, and the main focus of this work. It can verify programs written in Java, OpenCL, C, and PVL.

Most of these languages are well-known, except for PVL, which is the custom verification language for VerCors. It can be used to model concurrent programs in languages that do not have a frontend yet. As it is also intended as a direct interface for the backend, it can be used for the basic debugging of features.

VerCors is a deductive verifier. This means it applies the approach of deductive logic: it starts from a set of premises, and from those premises it tries to prove a logical conclusion using a logical system or a set of axioms. Deductive logic and verification is further explained in Chapter 3.

VerCors does modular verification. This means that verification of each method only depends on the contract of other methods. Therefore, the correctness of a method does not depend on the internals of any methods it calls: it only relies on the contract of called methods. This has the property that it scales well. Once a method `M` is proven correct, and other methods do not change their contracts, the method will remain correct, even if other methods change their implementations. However, if contracts of other methods change, method `M` will also need to be reverified. In particular, modular verification allows several techniques to be applied in a straightforward manner, such as caching of verification results, or

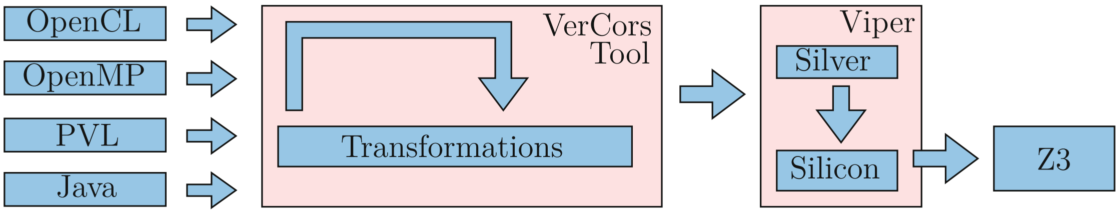


Figure 5.1: The architecture of the VerCors tool.

parallelization of method proofs. This modularity also extends to threads: if one thread is proven correct, this cannot be affected by other threads. That is, any thread can be added, and the initial thread will still be proven correct. This is also referred to as "thread-modular verification".

5.1.1 Architecture

The architecture of VerCors takes a classical approach of structuring a program analyser, where a series of transformations is applied to an internal representation.

When a program is verified, first the input is parsed and translated into Common Object Language (COL), the internal representation of VerCors.

Then, depending on the input language and the flags given, many passes are applied in sequence to simplify the program and abstract away from the input language. Examples of such passes are the `flatten` pass, which is responsible for flattening expressions, or the `pvl-encode` pass, which encodes PVL primitives for forking and joining as primitive asserts and assumes.

When the program reaches the end of the pipeline it is fully simplified and the last pass can be applied. This last pass converts simplified COL into the representation of the backend. The backend is then applied to the converted program, and usually returns a pass, fail, or unknown result. Any errors reported by the backend, such as failed asserts or preconditions, are translated back to the syntax level of the input and reported back to the user.

5.1.2 Example usage

To illustrate how an end-user would use VerCors to verify properties of their code, we discuss two examples of verification using VerCors. The first example will be an introductory example of sequential verification in Java. The second example will show a basic example usage of VerCors support for concurrency in PVL, as PVL allows succinct modelling of multiple threads.

Verification of sequential code

First, we show how to verify a simple sequential piece of toy code. In Listing 5.1 a method is presented that is supposed to calculate the maximum of a pair of integers.

Listing 5.1: A faulty implementation of a method computing the maximum of two integers.

```java
class C {
  //@ ensures (a > b ? a : b) == \result;
  int max(int a, int b) {
    if (a < b) {
      return a;
    } else {
      return b;
    }
  }
}
```

We have provided both a specification and an implementation. The implementation can then be verified against the specification:

Listing 5.2: `vct` is the VerCors executable. `--silicon` indicates that we want to use the Silicon backend.

```
$ vct --silicon sequential_example.java
Errors! (1)
=== sequential_example.java        ===
3   int max(int a, int b) {
4     if (a < b) {
       [---------
5       return a;
       ---------]
6     } else {
7       return b;
-----------------------------------------
AssertFailed:AssertionFalse
=========================================
=== sequential_example.java        ===
1 class C {
                [--------------------------
2   //@ ensures (a > b ? a : b) == \result;
                --------------------------]
3   int max(int a, int b) {
4     if (a < b) {
-----------------------------------------
caused by
=========================================
The final verdict is Fail
```

It looks like this code contains a bug, as it does not adhere to the specification. To fix it, we swap `return a;` with `return b;`. Then the code is verified again:

Listing 5.3: VerCors indicates the implementation adheres to the specification.

```
$ vct-dev --silicon sequential_example_fixed.java
Success!
The final verdict is Pass
```

Verification of concurrent code

In the next toy example we will show how to verify code with concurrency primitives. For this example we will use PVL, as expressing concurrency in PVL is concise.

The program in Listing 5.4 is supposed to add two values to the field `total`. To speed up matters, the work is divided over two threads. This is done by the `par` statement that executes any `{}` blocks that follow it in parallel.

Listing 5.4: A concurrent example program employing two threads to add two numbers to a shared field.

```
class C {
  int total;

  void add2(int amount1, int amount2) {
    par {
      total = total + amount1;
    } and {
      total = total + amount2;
    }
  }
}
```

VerCors can determine that there is a data race in this program without a specification:

Listing 5.5: Verification output of the example in Listing 5.4. For brevity only the first error is shown.

```
$ vct --silicon concurrent_example.pvl
Errors! (2)
=== concurrent_example.pvl          ===
4   void add2(int amount1, int amount2) {
5     par {
                [---------------
6       total = total + amount1;
                 ---------------]
7     } and {
8       total = total + amount2;
```

```
-----------------------------------------
AssignmentFailed:InsufficientPermission
=========================================
===  concurrent_example.pvl         ===
4   void add2(int amount1, int amount2) {
5     par {
               [-----
6       total = total + amount1;
                -----]
7     } and {
8       total = total + amount2;
-----------------------------------------
caused by
=========================================
...omitted...
```

The reason for this error is because we have not specified the permissions for `total`, as discussed in Chapter 4. This is an often recurring problem when verifying with VerCors, as specifying correct permissions is equal to proving data-race freedom of the program. Specifying permissions poses a challenge here: we could give both threads `1\2` permission, but that would only delay the problem as both threads want to write. Specifying `write` permission for both threads would be impossible as it would require two `write` permissions for `total`, which is impossible. The proper solution here is to add locks around the assignment and supply a `lock_invariant`, which specifies the resources that are protected by the lock:

```
class C {
  int total;

  resource lock_invariant() = Perm(total, write);

  C() { } /* Minimally an empty constructor is needed
           for the lock_invariant. */

  void add2(int amount1, int amount2) {
    par {
      lock this;
      total = total + amount1;
      unlock this;
    } and {
      lock this;
      total = total + amount2;
      unlock this;
    }
  }
}
```

Now VerCors reports that the program is data race free.

```
$ vct --silicon concurrent_example_fixed.pvl
Success!
The final verdict is Pass
```

While the permissions error was fixed (and hence the possible data race), we removed almost all of the parallelism in the program. To reinstate the parallelism a refactoring is necessary. For example, each thread could have its own temporary location for adding numbers, to which the thread would then have full write permissions. After all the threads finish, these intermediate locations could then be added together. Since this requires more effort than just a `for` loop however, this approach is only useful in the context of big workloads.

Verifying functional correctness of the program would be the logical next step. However, we do not discuss this here as it is not directly relevant to this introduction.

5.1.3 Design guidelines

During this work the aim of VerCors was kept in mind: to support developers in writing reliable concurrent software [8]. Whenever a design decision allows multiple choices, this aim was used to pick the next step. This resulted in the following two guidelines.

First, avoid arbitrary limitations if possible. Problems in commercial software sometimes require solutions that are not elegant, for the sake of efficiency reasons or time constraints. Examples of this are loops with multiple exit points, or inheriting for the sake of reuse. While frowned upon, they are sometimes necessary.

Second, we want to allow verification of real programming patterns. If a pattern is often used in the industry, it should be possible to ensure it is used correctly.

These two guidelines aim to minimize the number of changes needed before a program can be analysed by VerCors, which increases the chance that the techniques implemented in this work are used.

5.2 Viper

This section discusses the Viper tool, the main back-end of VerCors, and Silver, the input language of Viper. These are discussed because each transformation in VerCors aims to simplify the COL AST such that after all transformations have been done, it is trivial to convert the COL AST into Silver. Therefore, Silver is the context in which verification of the input program is done.

Silver is the intermediate verification language of Viper. Intermediate verification languages (IVLs) ensure that front-ends such as VerCors do not have to deal with the specific encoding of variables, objects, and logic constructs. Since this is facilitated by the IVL, the front-end can focus on encoding the high-level constructs into a lower-level language that is in between the complexity of a high-level language and pure logic. In particular, Silver is a kind of dynamic C-like language, with first class support for verification and separation logic. We will first discuss the basic language features of Silver, followed by a brief discussion of methods in Silver. Finally, we will discuss the facilities for classes in Silver.

5.2.1 Basic language features

The Silver language is a minimal version of a typical object oriented language. It has several primitive types, such as integers and booleans. It also has several high level types, such as sequences and sets.

In terms of statements the Silver language is minimal. There are only basic statements available, such as `if`, `while`, `goto`, `assert` and assignment to variables. It does not have `return`: to return a value from a method, the value has to be assigned to the return variable. After assigning to the return variable, execution continues with the next statement.

One notable difference between Silver and a commercial language like Java is that expressions in Silver do not have side-effects. For example, to encode an arbitrary Java expression, the expression will have to be flattened into assignments into intermediate variables. The only side-effects that are allowed are direct variable assignment, and method calls.

Silver is not intended for execution. Among other things, it allows forall and exists (∀, ∃) quantifiers in arbitrary expressions. Except for a few specific cases, these sort of expressions are not checkable if executed.

In Listing 5.6 basic usage of Silver statements is shown. On line 6 a sequence of even numbers is constructed. This property is checked by the assert on line 7. The sequence is returned by assigning it to the return variable `l` on line 8.

Listing 5.6: Example of a Silver program using sequences and the forall quantifier.

```
1 method addToList(n: Int) returns (l: Seq[Int]) {
2     var x: Int
3     x := n * 2
4     assert x % 2 == 0
5     var y: Seq[Int]
6     y := Seq(x, 4)
7     assert (forall i: Int :: 0 <= i && i < y ==> y[i] % 2 == 0)
8     l := y
9 }
```

5.2.2 Methods

Silver has methods, but does not allow the user to overload these: any method must have a unique name. Methods are not defined on a class or object: they are independent definitions, and hence there is not a keyword similar to `this` in Java.

Contracts are part of the built-in support for verification in Silver. These contracts are checked at both the call site and in the implementation. Contrary to Java, method implementations are not required: if an implementation is left out, the contract can still be used. This can be used to model functionality provided by the runtime environment that cannot be implemented, not yet implemented methods or to leave out methods explicitly that should not be verified.

In Listing 5.7 a basic example is shown of a Silver program. The method `triple` has no implementation, because only the contract is needed to reason about the program.

Listing 5.7: Example of a toy Silver program that doubles and triples numbers.

```
method double(x: Int) returns (y: Int)
requires x > 0
ensures y == x + x
{
  y := 2 * x
}

method triple(x: Int) returns (z: Int)
requires x > 10
ensures z == x + x + x

method compute() {
  var myX: Int
  myX := 10
  myX := double(myX)
  myX := triple(myX)
  assert myX == 60
}
```

5.2.3 Classes

Viper does not have classes and structs. Instead, it has functionality that is more expressive with which classes or structs can be emulated. It requires that all fields must be defined globally. These fields can then be used on any reference. The caveat is that fields can only be accessed if proper permission is available: reading requires positive permission, writing requires a permission of one. In other words, permissions are used to turn “on” and “off” certain fields on specific references. The syntax for a permission is as follows: `acc(`*location*`)`. Listing 5.8 shows an example of using fields in Silver. While assignment to `r.y` fails, this is not because the field does not exist on `r`, but because there is no permission. If a `write` permission were present, the assignment would have succeeded.

Listing 5.8: Example usage of fields in Silver.

```
field x: Int
field y: Int

method assignToField(r: Ref)
requires acc(r.x)
{
    r.x := 3
    r.y := 4 // Fails: insufficient permission to access r.y
}
```

Because Silver does not have classes or structs, it also does not have constructors like in Java. It is the job of the frontend to encode these language features in Silver.

Chapter 6

State of the Art

This chapter gives an overview of Viper frontends, deductive verifiers, and other static verification tools and disciplines. An overview of all the tools discussed in this chapter is given in Table 6.1. The chapter is divided into four sections with the following subjects:

- Tools that apply the strategy of using Viper as one of the verifier backends.
- Deductive verifiers that verify a commercial language.
- Tools that have achieved major milestones or have state-of-the-art features not found in verifiers of commercial languages.
- The benefits and shortcomings of approaches radically different from deductive verification, and their relation to VerCors.

The main purpose of this chapter is to substantiate our observation that there are few static verifiers that can reason about a big enough subset of their input language such that they are practically usable, support practical language features such as inheritance and exceptions, and also support concurrency. The ones that do are Nagini and Verifast, discussed in Section 6.1 and Section 6.2 respectively. One tool deserving an honourable mention is jStar, which supports inheritance but not exceptions. It is discussed in Section 6.3.

The tools discussed have varying levels of support for concurrency. We categorize these levels into three categories:

- No support: concurrency is not a concern for the tool and code is assumed to be sequential.
- Implicit support: concurrency can be modelled and verified, but the tool provides little to no support for starting threads or computation processes.

- Full support: the tool can model concurrent processes and access to shared memory.

To illustrate, a tool with implicit support at the time of writing is Prusti [4]: it can model concurrency, but is not far enough developed yet to allow the user to start concurrent threads. In terms of complexity, having full support is often harder than implicit support, as starting a thread often requires at least some specification of the standard library of the language.

There are many static verifiers for commercial languages, and even more static verifiers in general. To keep this chapter compact and the discussion relevant, verifiers that reason about verification languages are mostly ignored.

Verification languages are languages that are often designed with a specific purpose, such as debugging a back-end or to have a clean formal semantics. An example of such a language is PVL, as supported by VerCors, which can be used to debug the back-end or to model not yet supported languages.

While static verifiers reasoning about verification languages can be challenging to design and implement, the focus of this work is on practical language features such as inheritance and exceptions. These features are usually not present in verification languages and therefore these verifiers are not included.

A notable exception to this is Why3, which has support for exceptions, albeit a fairly clean and mathematical version of exceptions. It is discussed in Section 6.3.

6.1 Viper frontends

There are currently several Viper frontends besides VerCors at various levels of functionality. These are Nagini, Prusti, Soothsharp, Rust2Viper and Scala2Sil.

The most notable Viper frontend at the time of writing is Nagini. Nagini [23] is a static verifier for Python developed at ETH Zürich. It supports Python 3 with mandatory type annotations as outlined in [57]. This includes inheritance, exceptions, and full support for concurrency. It also allows verification of finite blocking and input/output behaviour. From all the Viper frontends, Nagini is the most complete in that it supports almost the entire input language. The support of Nagini for exceptions and inheritance will be discussed further in Chapter 9. This is because it is the only checker that has the same level of support for exceptions and inheritance that we are trying to achieve.

Besides Nagini and VerCors, the other most actively developed Viper frontend is Prusti. Prusti [4] is a static verifier for Rust using Viper as a backend developed at ETH Zürich. It employs the key insight that type checked Rust programs already incorporate all the information necessary to derive permissions for reading and writing to variables. Therefore, in function contracts only functional properties have to be specified; permission properties and magic wands are derived

Name	Development	Viper	Concurrency	Exceptions	Inheritance
Nagini	Current	Yes	Full	Yes	Yes
Prusti	Current	Yes	Implicit	No	No
Soothsharp	Prototype	Yes	Implicit	No	No
Rust2Viper	Prototype	Yes	Implicit	No	No
Scala2Sil	Prototype	Yes	Implicit	No	No
Frama-C	Current	No	Full	No	No
Verifast	Current	No	Full	Up to `finally`	Yes
KeY	Current	No	No	Yes	Yes
OpenJML	Current	No	No	Yes	Yes
JaVerT	No	No	No	Yes	No
K	Current	No	Full	–	–
Spec#	No	No	No	Yes	Yes
jStar	No	No	Implicit	No	Yes
LOOP	No	No	No	Yes	Yes
Krakatoa	No	No	No	Yes	No
VCC	No	No	Full	–	–
Caper	Unclear	No	Implicit	–	–
Why3	Current	No	No	Yes	No

Table 6.1: This table summarizes all tools discussed in Chapter 9. The “Viper” column indicates if the tool is a Viper frontend. The “Development” column indicates if the tool is still being developed, with “Current” indicating it is, and “Prototype” indicating it was developed as an experimental prototype.

automatically. Rust features supported by Prusti are among others: primitive types, several compound types (enums, tuples, structs), functions, methods, and traits. Several features like unsafe code and closures are not yet supported but might be in the future. It currently only has implicit support for concurrency.

The next three Viper frontends are Soothsharp, Rust2Viper, and Scala2Sil. They are all prototypes resulting from Masters theses that are no longer being developed. They do not have full support for concurrency, but since they all employ separation logic they have implicit support for concurrency because they specify permission to fields through separation logic. While they are not as mature as the previously mentioned verifiers, they illustrate what is possible in the space of Viper frontends.

Soothsharp [34] is a C# verifier developed at the Charles University of Prague. Soothsharp uses Viper as a backend for its analysis. It supports various C# features such as permissions, overloading, arrays, and classes. However it currently still lacks features such as exceptions, concurrency and inheritance.

Rust2Viper [27] is a Rust verifier developed at ETH Zürich. It uses Viper as its backend for analysis and supports various features such as basic control flow, borrow checking, enums and structs. It does currently not support traits nor concurrency. While the lineage is unclear, it seems that Rust2Viper is a predecessor to Prusti, as they share some syntax but Prusti has more features.

Brodowsky developed a Scala verifier which from here on will be referred to as Scala2SIL [10]. It uses Viper as its program analysis backend and supports a significant subset of Scala features, such as subtyping, advanced type system features, permissions and classes. However it currently still lacks features such as exceptions, concurrency, and inheritance.

6.2 Deductive verifiers for commercial languages

In this category, first four well-known mature checkers will be discussed: Frama-C, Verifast, KeY, and OpenJML. They were selected because they are mature tools, and give a good indication of what is the state of the art for deductive verifiers. Then two more recent experimental verifiers will be discussed, namely JaVerT and the K semantic framework. These were selected to show that there are also tools that verify dynamic languages, as well as tools that do not follow the prevalent "frontend-backend" architecture.

The Frama-C [43] analyser is a platform for static analysis of C code. It has various modules for analysing C code. Among others, the "Value" plugin does forward dataflow analysis, the "WP" plugin generates verification conditions using a weakest precondition calculus, and the "Mthread" plugin extends the "Value" plugin to concurrent environments. The "Mthread" plugin computes which variables are shared safely and which variables are prone to data races. Therefore, Frama-C has full support for concurrency in the sense that it can detect data-races and reason about if mutexes protect a shared memory region properly. However, it does not have explicit support for modelling concurrency besides the use of locks in the code. Furthermore, Mthread can also output all interleavings between threads, showing which instructions might lead to a runtime error. While powerful, this is not thread-modular (as defined in Section 5.1), which can be considered a downside.

The Verifast [65] tool is a static verifier for Java with full support for concurrency. It is not explicitly stated what the supported Java features are. However, from the examples included with Verifast it can be concluded that it is a subset of Java SE 7. Verifast facilitates inheritance through abstract predicate families, which this work also uses and are further discussed in Chapter 4. For managing access to resources in the context of concurrency and threads it uses separation logic. It verifies programs by using the symbolic execution algorithm for separation logic outlined in [7]. While Verifast and VerCors are both deductive verifiers, their modes of use are quite different. Verifast is not so much focused on automation of program verification, but more on expressiveness and debugging [40]. VerCors also allows this style, but endeavours for more automation of the proving process and making formal verification of software usable for practical languages [8].

KeY is another static checker for Java. It supports the sequential subset of Java Card and Java 1.4 (released in 2002). This means KeY can reason about exceptions, inheritance, dynamic dispatch, strings, arithmetic and more [1, 42]. KeY supports some extra modern features like enhanced for loops and compile-time removal of generics [42]. While old, it is still frequently used by researchers and teachers, such as [29] and [2]. The tool is quite mature. Its interface is different from VerCors in the sense that KeY is more similar to a (automated) theorem prover. When a proof is being constructed for a program and KeY gets stuck the user is presented with the intermediate state of the proof. The user can then select proof rules and axioms to apply to the intermediate state, hopefully simplifying or splitting up the state, such that KeY can continue again on its own.

One more checker targeting sequential Java is OpenJML [14]. It supports a subset of Java 7, including exceptions, inheritance, and partial support for generics. It is intended as a successor of ESC/Java and ESC/Java2. It improves upon ESC/Java2 by removing sources of unsoundness and incompleteness and supporting a bigger subset of Java.

The past 3 years, checkers for dynamic languages have also been getting more attention. JaVerT [24] (short for "Javascript Verification Toolchain") is a static verifier for sequential JavaScript developed at the Imperial College of London. They target ECMAScript 5 "strict mode", which means they support prototypal inheritance, exceptions and function closures, among other JavaScript language features. JaVerT converts JavaScript into its own Javascript Intermediate Language (JSIL), which is symbolically executed to generate proof obligations, which are discharged through Z3. They implement their own solver for separation logic assertions.

While prototypal inheritance can be used to model classical object oriented inheritance, it is not the same. JaVerT allows the user to define a class using the prototypal inheritance features of JavaScript. The user can then specify a contract for the methods in the prototype. However, to the best of our knowledge, JaVerT does not provide a mechanism such as specification inheritance, abstract predicate families, or static/dynamic contracts (all discussed in Chapter 8), to subclass and specialize JavaScript classes based on other classes. It does allow prototypes to be switched out for other prototypes, but the contracts of these prototypes must be exactly the same. This results in a system more limited than what inheritance in Java allows in general.

And lastly, there is the approach of the K semantic framework of generating static verifiers from a programming language semantics. It is developed by the University of Illinois and the Alexandru Ioan Cuza University of Iaşi. It can already generate verifiers for Java, C and JavaScript [67]. K builds on a theory of a language-independent proof system. This combined with several language-independent axioms and a separate programming language semantics makes it possible to generate verifiers. At the core of K is “reachability logic”, which replaces the often used Hoare logic in static verifiers. Reachability logic allows to directly encode the executable semantics of a programming language into an operational semantics. Ştefănescu et al. have extended K with support for concurrent semantics [66]. The direction of this research is promising. However, work still remains to be done to make this approach practically usable as generated verifiers are relatively slow. Furthermore, the K semantic framework can only generate model checkers and interpreters, but not yet deductive verifiers. Work on adding this functionality is ongoing.

6.3 Milestone tools

Over the years a few well known tools have been decommissioned: Spec#, jStar, LOOP, Krakatoa and VCC. These were notable checkers that achieved major milestones such as being integrated into a well-known IDE, were the first to formally verify important design patterns, or supported a substantial subset of the input language. They will be discussed first, then Caper will be discussed, which implements static verification of “fine-grained concurrency”. Lastly, Why3 will be discussed, a verification backend with built-in support for exceptions.

Spec# is a verifier developed at Microsoft Research [5]. It can verify a sequential superset of C#, including inheritance, exceptions, and frame conditions. It can verify the specified contracts both statically and dynamically. Spec# uses Boogie as its solver backend. When Spec# was still in active use, it was recommended to use the Visual Studio plugin when developers wanted to use it.

jStar [20] is a verifier developed by Queen Mary University of London, Microsoft Research Cambridge, and Cambridge University. It verifies Java and supports inheritance through abstract predicate families, but not exceptions. It is not stated explicitly if jStar targets sequential or concurrent Java, but since it uses separation logic it at least has implicit support for concurrency. It allows encoding of ownership through separation logic, and can infer some loop invariants through a fix-point algorithm. The supported cases can be extended by using user-supplied "abstraction rules". jStar contains a symbolic execution module for generating verification conditions and an embedded theorem prover that discharges the verification conditions. jStar can verify real object-oriented programming patterns such as the visitor pattern, the factory pattern, and the pooling pattern [20].

LOOP [39] is a verifier developed at Radboud University Nijmegen. It verifies sequential Java, including exceptions and inheritance. It supports a complete subset of Java: static/non-static fields, methods and overloading of these methods. Jacobs and Poll claim the only major features not supported are threads and inner classes. The approach of LOOP was to compile Java programs with JML annotations into a shallow embedding in PVS. The verification of the program and the specification thus depended on proving it correct in PVS. The benefit of this approach is that it gives the user the full power of an interactive theorem prover to prove the program correct. However, the drawback is that there is a barrier for using LOOP: the user needs to be skilled at both JML and PVS to be productive.

Krakatoa [47] is a verifier developed at INRIA. It verifies sequential Java and has support for exceptions. It does not support inheritance, but attempts have been made to incorporate this functionality [21]. Krakatoa uses the Why verification platform and Coq as back-ends. The general approach of Krakatoa is to transform a Java program into a corresponding representation in WhyML, the custom input language of the Why verification platform. Why is then used to generate proof obligations in Coq, an interactive theorem prover. If these proof obligations can then be proven in Coq, it can be assumed the original Java programs respected their specifications. By taking this approach it was similar to LOOP, as it depended on an interactive theorem prover to prove correctness. Marché, Paulin-Mohring and Urbain have focused primarily on this approach (specifically in the context of Why 2.41), but we expect that with the developments of Why3 some proof obligations that originally required Coq can now be proven by Why3 itself.

Verifying Concurrent C (VCC) [13] is a static verifier originally developed by

Microsoft. It is no longer under development. VCC can verify concurrent C code and also has support for x86 assembly, which often occurs inline in critical C code. It uses Boogie [45] as the backend prover. Among other features, VCC allows the user to express constraints using typestates, ghost state, and frame conditions. The primary goal of VCC was verification of the Microsoft Hyper-V hypervisor. [13] reports that at least 20% of the Hyper-V codebase has been formally verified with VCC.

Caper [18] is a static verification tool for verifying fine-grained concurrent programs. "Fine-grained" in this context implies that write access to a single resource can be shared between multiple process. To paraphrase an example from [18]: given an integer variable x, theoretically one could assign "increment permission" to one process and "decrement permission" to another. Using these permissions, both processes could infer an upper or lower bound of x respectively during their lifetime, even though they both have write access to the variable. Caper allows specifying these kinds of constraints and verifying them. Caper applies "region aware" symbolic execution and separation logic to verify data structures such as a spin-lock, ticket lock, and a stack-based bag. It has a custom input language for describing these data structures.

Why3 [9] is a verification environment similar to Viper. It verifies WhyML (Why Meta Language), which is intended as an intermediate language or back-end for other verification tools. Why3 has been used as a back-end in Frama-C, Krakatoa, and Spark2014. It has support for many back-ends that can discharge verification conditions: CVC4 (Cooperating Validity Checker 4), Coq, Z3, Isabelle, and others. It does not support concurrency, however, parallelism can be modelled on top of it. For instance, Santos, Martins and Vasconcelos modelled a protocol on top of Why3 and subsequently proved the protocol to be free of deadlocks [63]. They did this by implementing an MPI-like interface, and then transforming the protocols to Why3 code using the MPI interface.

Besides WhyML, Why3 can also verify Micro-C and Micro-Python, subsets of the original languages. We still include it in this overview because it has support for exceptions. The model of exceptions supported is clean. There is no `finally`, no subtyping when catching exception types, no checked/unchecked distinction, or in short: no practical warts. It more or less allows the user to define multiple return types for a function, discriminated by the types the functions can throw, combined with the ability to return early. Furthermore, exception types can only be used when raising an exception, catching an exception, or defining the types thrown by a function, and not for other purposes such as variable types.

6.4 Other approaches to static verification

Besides the deductive verification approach as implemented by the programs above there are also two other approaches to static verification of software: model checking and interactive theorem proving. This section describes these two approaches and discusses how they differ from deductive verification.

Model checking Model checking is used when the safety and liveness requirements of an application are strict. It allows the user to model the entire state space of their programs and specify functional invariants and safety/liveness constraints that have to hold at various points in the program. A model checker can then traverse the specified state space and compute whether or not these constraints hold. To remain efficient, model checkers often have optimizations to reduce this state space.

Model checking is useful if a property needs to be checked across all the states of a system, or if it needs to be checked that a certain state is unreachable. Conversely, if a property needs to be proven after a certain point in the program, or an implication needs to be proven, deductive verification is often a better fit.

Representative examples of model checkers are NuSMV [12], Spin [31] and UPPAAL [17].

Interactive theorem proving Interactive theorem provers (ITPs) are useful when proving the functional correctness of a system. This system is usually a logic or language, but can also be a model of a concrete physical system.

ITPs allow the user to specify formal structures and functions, and to construct theorems out of these formal elements. The ITP can then try to prove these theorems automatically using various approaches, often referred to as "tactics". These tactics apply rewriting rules or proof steps specific for that tactic. When a tactic cannot automatically prove a lemma, the user can try a different tactic or apply proof steps manually.

The "interactive" part in ITP refers to the explorative and iterative nature of ITPs. Sometimes an ITP can prove a theorem in a single go. Other times an ITP gets stuck, and the user has to try different tactics to get the ITP to continue. If everything fails, a user can choose to try and either apply manual proof steps, or prove intermediate lemmas the ITP is missing, such as associativity or commutativity of an operation.

In theory, ITPs could be used for verification of commercial languages like Java and C. Unfortunately ITPs are currently not a good fit for software verification because their interface is intended for logicians and mathematicians. The explorative nature of ITPs, strange syntax, and a steep learning curve make it less attractive for software developers in general. However, there are efforts to challenge this status quo, such as [44], which tries to bridge the gap between ITPs and commercial language tooling.

Representative examples in this category are Isabelle [48], Coq [11], and PVS [15].

Chapter 7

Exceptions

This chapter is about exceptions and their implementation for Java in VerCors. First the design considerations of support for exceptions are discussed, as well as three candidate encodings in Silver. Then the concrete implementation in VerCors is discussed, followed by an evaluation.

7.1 Design considerations

In this section we discuss design considerations for the exception transformation. We start by defining the problem, and outlining what aspects of exceptions are considered and what not. We then discuss the concurrent aspects of exceptions, and then discuss the semantics of exceptions used in the transformation. Finally, we discuss why `finally` makes encoding exceptions in Silver difficult, and we compare several candidate encodings.

7.1.1 Problem statement

Verification of exceptions can be interpreted in different ways. Therefore it is useful to define how it is approached in this work.

Verification of exceptions can be interpreted as formal reasoning about language features in the context of exceptions. This translates into supporting the various keywords that relate to exceptions: `throw`, `try-catch-finally`, `signals`, and others.

However, verification of exceptions can also be interpreted as formal reasoning of standard library features in the context of exceptions. This is related to language features but slightly different in the sense that it focuses more on actual specifications. The most interesting example of this is `Thread.UncaughtExceptionHandler`. This is the handler that is called when an exception is not handled in a `Thread`.

Listing 7.1: Example of conditional permission because of allocation within `if`. The field `f` is only available if the `if` block is executed, and hence dependant on the value of `p`.

```
void foo() {
  MyObject o;
  if (p) {
    o = new MyObject();
  }
  //@ assert p ==> Perm(o.f, 1\1);
}
```

This terminates the thread, which often conceptually translates into a failure or loss of work. The user is responsible for using this `UncaughtExceptionHandler` and recovering gracefully from a failure. It is challenging to specify this property, however progress in this area is being made by works such as ‘Provably Live Exception Handling’ [37].

Other examples of verification of standard library features related to exceptions are `NumberFormatException` as thrown by `Integer.parseInt`, or `InterruptedException` as triggered by `Thread.interrupt`.

This work focuses primarily on the first interpretation of verification of exceptions: to implement support for the primitives that are used when developing programs that throw and catch exceptions.

7.1.2 Concurrent considerations

This work focuses on verification of concurrent programs. To the best of our knowledge, for exceptions there do not seem to be problems in terms of concurrency. However, the combination of separation logic and exceptions might cause practical usability problems.

One example of this are conditional permissions. These are permission that are only available if a certain condition is met. An example of such a conditional permission is presented in Listing 7.1.

Such permissions are not problematic for Viper to work with, as they are well defined. However, they might lead to unclear or verbose specifications because permissions are only usable once certain conditions have been met. In the example, this condition is only `p`, but in a more complex example this could involve multiple variables and methods. Hence, this is purely a usability and interface issue, and less a technical or theoretical issue (unless conditional permissions are bad for verification performance - but we have not seen evidence for this).

It is possible that exceptions cause such conditional permissions to occur more often. For example, if an exceptions can be thrown halfway through a `try` block, the state after the `try` block depends on whether the exception was thrown or not.

Solutions can already be developed that limit occurrence of such conditional permissions. For example, an "exceptional invariant" can be imagined that has to hold both at entry and exit of a `try-catch` block, as well as at the end of `catch` clauses. However, in our opinion more concrete experience with exceptions and separation logic is needed to find out if usability of exceptions is problematic in the long term, as that experience is currently lacking. If more notational convenience is needed for exceptions in separation logic, an informed solution can be designed based on detailed knowledge of the problem.

7.1.3 Exception semantics

Before a transformation from exceptions to Silver can be defined, first the semantics of exceptions as verified by VerCors must be defined. Specifically, it must be defined what the sources of exceptions are in Java.

If *The Java language specification* [41] is interpreted to the letter exceptions can originate from many places. Some examples include but are not limited to:

- Class loads can trigger `OutOfMemoryError`,
- The increment operator (`i++`) can trigger `OutOfMemoryError` if autoboxing is required,
- `Array.clone` can throw `InternalError` if the element type is not cloneable
- and `new` can trigger `OutOfMemoryError`.

These kinds of exceptions are called "spurious exceptions": exceptions that can occur almost anywhere. Note that spurious exceptions cannot occur randomly. Instead, it is the case that the places where exceptions can come from are many. The annotation burden on the user would be enormous, and hence we cannot require the user to annotate for these exceptions as well.

To avoid this problem and remain productive this work recommends a "best effort" approach where VerCors reasons about a constrained but useful subset of exceptions. Informally, the approach implemented in this work allows verification exceptions thrown by `throw` and method calls. We think this will not cause major problems in the foreseeable future, as most developers only consider these two sources of exceptions to begin with.

Formally, if VerCors does not report any errors when verifying a program it implies the following guarantee:

For all methods, if an exception is thrown (originating from either a `throw` or a throwing method call), it is either handled in a surrounding `catch`, or the method declares the type in a `signals` or a `throws` clause. In addition, several designated exceptional cases can also not occur.

At the time of writing "several designated exceptional cases" include:

- `NullPointerException` when a `null` reference is dereferenced.
- `ArithmeticException` when division by zero or modulo zero takes place.
- `ArrayIndexOutOfBoundsException` for out of bounds array accesses.

VerCors disallows these situations to occur, so the exceptions also cannot occur by default. However, nothing prevents this set from being extended with static detection of other exceptional situations.

Practically speaking, this guarantee implies that all declared exceptions originating from most practical places where you would expect them are handled with appropriate `catch` clauses. We leave annotation and verification of spurious exceptions for future work.

7.1.4 The `finally` encoding problem

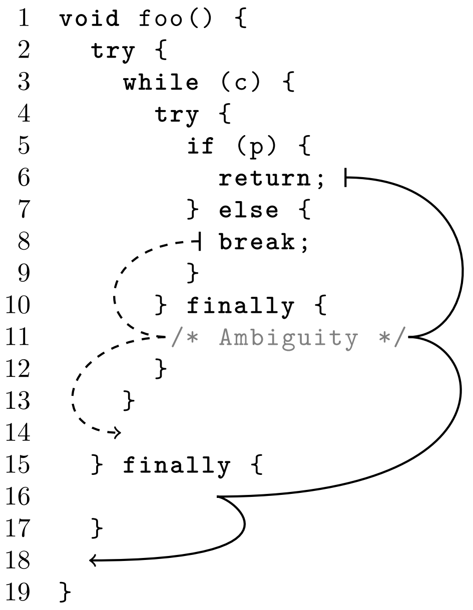
If `finally` is not considered, encoding abrupt termination into `goto` is straightforward. This is because the description of the semantics as given in [41] can be interpreted literally. An overview of the transformation is as follows:

- `throw` jumps to the nearest handler, or to the end of the function if there is no handler.
- After a `catch` clause execution continues after the `try`.
- `break` jumps to after the nearest loop.
- `return` jumps to the end of a function.
- If a method call throws it either jumps to the nearest handler or to the end of the method.
- After a `try` block execution should continue after it.

However, when `finally` is introduced, a more intricate transformation is needed. This is because contrary to all other abrupt termination primitives, at the end of a `finally` clause it is not directly clear where to jump to.

Consider the example in Listing 7.2. The lines drawn indicate how control flow would progress. With the control flow explicitly drawn, reasoning about the control flow is easy. However, without the lines it is less clear what exactly must happen on line 11. If `break` was just executed, control flow needs to to jump to after the `while` on line 14. If `return` was just executed, control flow needs to

Listing 7.2: Example of ambiguous control flow in the presence of `finally`. Control flow from `break` and `return`is made explicit through arrows.



```
1  void foo() {
2    try {
3      while (c) {
4        try {
5          if (p) {
6            return;
7          } else {
8            break;
9          }
10       } finally {
11         /* Ambiguity */
12       }
13     }
14
15   } finally {
16
17   }
18
19 }
```

jump to the next `finally` on line 16. Without any further information, there is an ambiguity on line 11 which can only be resolved by knowing what kind of statement was just executed.

Therefore to encode `finally` blocks, what "kind" (returning, breaking, or throwing) of control flow currently applies needs to be encoded. Furthermore, once labelled breaks are added to the language it becomes even more complicated since which *specific* loop is to be broken out of also needs to be tracked. An example of this is shown in Listing 7.3, where again all code paths from abrupt termination are drawn. Whenever a labelled break is used within a `try-finally`, additional jump destinations will have to be managed.

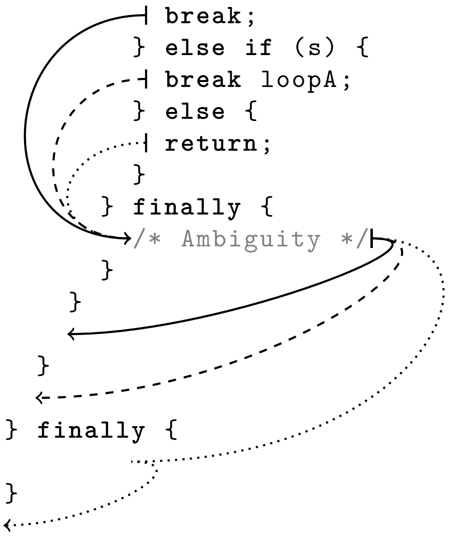
We conclude that if `finally` is used in a method, extra measures need to be taken to encode all possible jumps correctly.

7.1.5 Candidate encodings

Several candidate encodings for `finally` and the rest of the abrupt termination primitives are possible. We discuss the three encodings known to us next. The chosen encoding (via exceptions) is outlined in detail in Section 7.2.

While the first and second of these encodings have appeared in some form in

Listing 7.3: Example code showing different code paths when a labelled break is added.



```
void foo() {
  try {
    loopA: while (p) {
      while (q) {
        try {
          if (r) {
            break;
          } else if (s) {
            break loopA;
          } else {
            return;
          }
        } finally {
          /* Ambiguity */
        }
      }

    }

  } finally {

  }

}
```

an implementation before this work, we have not yet seen an effort to categorize and compare the approaches.

Inlining The first option that comes to mind is to inline all `finally` blocks in places where normally control flow would jump to the next place of interest. For example, before a throwing method call would jump to a handler, the `finally` clause could be executed by inlining it right there.

This option is interesting because it is conceptually straightforward. It is also used in Java compilers [28, p. 3], informally showing that the approach works.

The downside of this encoding is code duplication. First, it is bad for VerCors, since it will blow up the amount of memory needed to store an AST with encoded `finally` clauses. Even though this blow-up is minimal, as empirically shown in [25], this is still wasted memory. Second, it is bad for the prover backend, as duplicated code might cause duplicate proof obligations, which in turn will increase the time needed to prove the program correct. We have performed a small experiment that shows that this is indeed the case for VerCors. This experiment is discussed in Appendix B.

Control flow flags The second option is the optimized version of the first option: `finally` blocks are not inlined, but instead a flag is set whenever the mode of control flow changes. For example, when a `return` is executed, the flag is set to a constant called `MODE_RETURN`. This flag can then be queried at the end of a `finally` clause to determine where next to jump to. There should be values for each available mode of abrupt termination (i.e. `break`, `return`, `throw`), as well as a mode for every label that can be broken from.

As far as we can tell this is technically possible, but getting the bookkeeping right and keeping track of all the labels and modes available seems difficult. Furthermore, at the end of every `finally` clause there will be a big `if` statement determining where to jump next, unless the possible labels at that point are pruned in a smart way. This big `if` statement again is tricky to get right because it is non-modular and needs information from other places in the program.

One example of a verifier that uses this approach is Nagini, as discussed in Section 9.1.

Another approach to encode `finally` into `goto`'s similar to the control flow flag approach is presented in [25]. When that paper was published the JVM used to encode `finally` blocks as JVM subroutines, which are routines local to the method that do not allocate extra stack space. Freund suggests `finally` can instead be encoded as a code segment that is jumped to and from using `goto`. Additionally, when jumping to this block an extra value has to be supplied in an auxiliary variable that indicates where the finally block must jump to when it is label `l_S` is fresh

```
010: call S
011: ...
030: call S
031: ...
100: subroutine S:
101:  ...
120:  subroutine return
```

$\Longrightarrow$

```
009: push 0
010: goto l_S
011: ...
029: push 1
030: goto l_S
031: ...
100: label l_S:
101:  ...
120:  pop into r1
121:  if r1 == 0 goto 011
122:  if r1 == 1 goto 031
123:  goto panic
```

Rewrite rule 1: Transform subroutine to `goto`. In pseudo-assembly.

finished. An abbreviated version of this transformation in pseudo-assembly can be found in Rewrite rule 1.

While this approach succeeds in avoiding duplicating code needlessly, it contains the same downside as the control flow flags approach. Each `finally` must communicate its flags and values to the call sites, and each `finally` must also know about all the sites it is called from, so it can jump back to them afterwards. Similar to the control flow flags approach this results in an `if` statement at the end of the `finally` statement that introduces complexity at other places in the program.

Via exceptions The third option is to consider abrupt termination from an exceptional point of view. When only exceptional control flow is considered the question of where to continue at the end of a `finally` clause is simplified:

- If an exception is currently being thrown, execution should continue at the next `catch` clause. If there is no such clause, either go to the next `finally`, or otherwise to the end of the function.
- If an exception is not being thrown, continue after the `try-finally` block.

Note that the choice of where to jump at the end of the `finally` clause has become more modular: it does not matter how many exceptions or labels are in play, as long as the next `finally` or `catch` clause is known. By homogenizing control flow into the exceptional model, the choice at the end of a `finally` clause becomes straightforward.

A downside of this encoding is the requirement for this simplification to apply: all other abrupt termination must be removed or transformed into exceptional type `l_Ex` is fresh

```
                                    class l_Ex
                                      extends Throwable
                                      {}
l: while (b) {                      ...
  ...                               try {
  break l;                           while (b) {
  ...                    ==>           ...
}                                      throw new l_Ex();
                                       ...
                                     }
                                    } catch (l_Ex e) {}
```

Rewrite rule 2: Transform `break` to `throw`.

control flow. This is extra work, but we argue that it is not difficult. An example of how `break` can be encoded as `throw` can be seen in Rewrite rule 2. The translation is similar for labeled statements in general and `return`. This combined with the fact that it leads to a more straightforward encoding leads to the choice of implementing this encoding in VerCors. Additionally, if `finally` is not present, the basic encoding into `goto` can be used.

A semantic downside of compiling to exceptions is that information is lost. Since all control flow is exceptional after the transformation, exceptional control flow is the rule and not the exception. We do not believe this is a fundamental problem. If this information is needed it can be encoded in the AST. This ensures that synthetic `try-catch` and `throw` can be discerned from authentic ones. By adding an extra flag even the current control flow can be identified as synthetic exceptional or natural exceptional. However, it does introduce extra work and makes the transformation less elegant, whereas the other approaches preserve this information syntactically by default.

While we discovered this encoding independently, a verifier that uses a comparable approach is Krakatoa. We discuss the differences with our encoding in Section 9.7.

7.2 Transformation of abrupt termination

This section discusses the transformation of a language with abrupt termination and exceptions to a language without. First a brief overview is given of the parts of the transformation, and then each part is discussed in detail. The transformation closely mirrors the implementation in VerCors.

The transformation consists of three distinct phases. There is the pre-processing phase, the abrupt termination phase, and the post-processing phase. The phases are outlined in Figure 7.1.

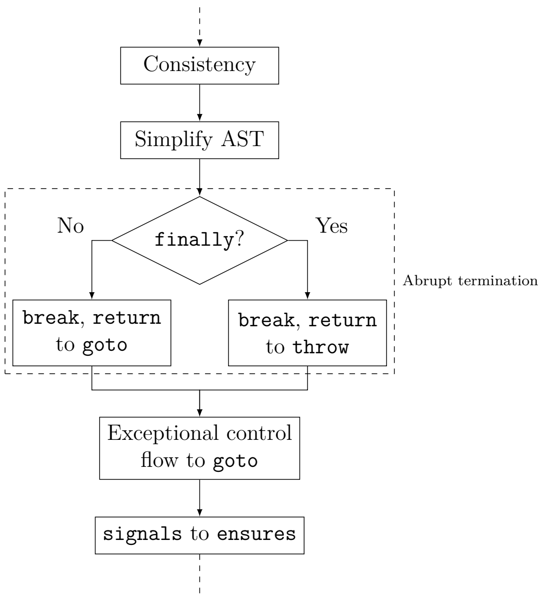


Figure 7.1: Summary of the transformation of a language with exceptions and `finally` to a language without exceptions and `finally`.

The pre-processing phase is responsible for doing a consistency check and standardizing the AST. The consistency check consists of type checking and checking for correct use of Java language features, such as using existing labels for `break`s. Since this is not relevant for this work, it is not discussed. Standardizing the AST is discussed in Section 7.2.1.

Then the abrupt termination phase follows. Here it is decided if abrupt termination can be encoded as `goto` or as `throw`, depending on the use of `finally` in the program. This is purely a cosmetic decision: in our opinion the `goto` encoding produces slightly more readable Silver programs. The `goto` and `throw` encodings are discussed in Section 7.2.2 and Section 7.2.3 respectively.

After the abrupt termination phase, all abrupt termination primitives are either encoded as `goto` or encoded as `throws`. Only exceptional control flow and `signals` need to be encoded in this phase. The steps involved in this phase are described in Section 7.2.4 and Section 7.2.5 respectively.

The transformation steps described in the next sections include rewrite rules. These rewrite rules indicate visually what parts of the AST are shifted around to achieve the goal of the transformation step. They mimic proof rules in their appearance, but are not as formal. In these rewrite rules, such as Rewrite rule 4 on the next page, above the line any conditions for the rewrite to take place are listed. Below the line, there are two listings. The left listing shows what pattern must be matched for the rule to apply. The right listing shows what program the match should be replaced with. Triple dots (...) are used as wildcards.

7.2.1 Simplifying AST

Before abrupt termination can be encoded into `goto`, it must first be ensured that the AST is in a minimal and consistent form. This is done through the following steps:

Encode `throws` as `signals` `throws` is encoded as a `signals` clause with the default post-condition `true` if there is not yet a `signals` clause present for that type. This means that after this step, all types a method can throw can be derived from the `signals` clauses. This also means the `throws` attribute can be discarded. See Rewrite rule 3.

$$\frac{\texttt{E}\text{ does not yet occur in }\texttt{signals}}{}$$

```
void m() throws E;
```

$\Longrightarrow$

```
//@ signals (E e) true;
void m();
```

Rewrite rule 3: Encode `throws` as `signals`

Add implicit labels Unlabelled `break`s and `continue`s require analysis of the surrounding code to know where the control flow will continue. Making these implicit labels explicit simplifies later transformation steps. Additionally, this step also ensures the added labels are unique. See Rewrite rule 4.

$$\frac{\texttt{l}\text{ is fresh}\quad \mathit{target}(\texttt{break}) = \texttt{l}}{}$$

```
while (b) {
  ... break; ...
}
```

$\Longrightarrow$

```
l: while (b) {
  ... break l; ...
}
```

Rewrite rule 4: Label `while` statements. `l` can be chosen freely, as long as it is unused. This rule only matches if the `break` breaks from the matched `while` loop, and not some other `while` loop.

Encode `continue` as `break` `continue` can be encoded as `break` without loss of semantics. This has the benefit that later code does not have to consider `continue`, but only `break`, `return` and `throw`. See Rewrite rule 5.

$$\frac{\textit{true}}{\quad}$$

```
                                      l: while (b) {
l: while (b) {                          inner_l: {
  ... continue l; ...                     ... break inner_l; ...
}                            ⟹          }
                                      }
```

Rewrite rule 5: `continue` to `break`

7.2.2 break, return to goto

If `finally` is not used, `break` and `return` can be encoded into `goto`. This is achieved through the following steps:

Encode `break` `break` is encoded as `goto`. This is done by emitting an `after_l` label after every labeled statement with label `l` and replacing every `break l;` with `goto after_l;`. See Rewrite rule 6

$$\frac{\textit{true}}{\quad}$$

```
                                      while (b) {
l: while (b) {                          ... goto after_l; ...
  ... break l; ...                    }
}                            ⟹        after_l:
```

Rewrite rule 6: `break` to `goto` for `while`. This can be generalized to other compound statements with labels.

Encode `return` `return` is encoded by replacing a `return` with an assignment into a return variable named `result`, an assertion of the post-condition and a jump to the end of the function. At the end of the function, the return variable `result` is returned. This is to allow later transformation steps to pick a different way of encoding the return value if needed. See Rewrite rule 7.

true

```
//@ ensures p;
T m() {
  ... return e; ...
}
```

$\Longrightarrow$

```
//@ ensures p;
T m() {
  ...
  result = e;
  //@ assert p;
  goto m_end;
  ...
  m_end:
  return result;
}
```

Rewrite rule 7: `return` to `goto`

7.2.3 break, return to throw

If `finally` is used, `break` and `return` are encoded as `throw`. This is achieved through the following steps:

Encode `return` as `throw` The method body must be wrapped in a `try-catch` block such that a "return exception" can be thrown to emulate control flow from `return`. Each `return` is replaced by a `throw` throwing the return exception. The type of this return exception depends on the name of the method. Additionally, if the method returns a value, this value is inserted in the exception through the constructor. See Rewrite rule 8.

true

```
T m() {
  ... return v; ...
}
```

$\Longrightarrow$

```
T m() {
  try {
    ...
    throw new Ret_m(v);
    ...
  } catch (Ret_m e) {
    return e.value;
  }
}
```

Rewrite rule 8: `return` to `throw`

Encode `break` as `throw` Any labeled statement that contains a `break` must be wrapped in a `try-catch`. Each `break` is then replaced by a `throw`. The type thrown and caught is named after the label of the labeled statement. See Rewrite rule 9.

```
                              true
                           ----------
                                 try {
                                   if (p) {
l: if (p) {                          ...
  ... break l; ...                   throw new Ex_l();
}                          ==>       ...
                                   }
                                 } catch (Ex_l e) { }
```

Rewrite rule 9: Rewrite `break` to `throw`. This rewrite rule can be generalized to other compound labeled statements such as `while` and `switch`.

7.2.4 Exceptional control flow to `goto`

At this point, `break` and `return` are encoded in `goto` or `throw`. This means the control flow in the AST strictly consists of `goto` or exceptional related control flow. The following steps encode exceptional control flow into `goto`.

Encode `try`, `catch`, `finally` Each `catch` clause gets an entry label so it can be targeted by `goto`s. Furthermore, at entry of the catch clause the `exc` variable is inspected. If it does not have the same type as the catch clause, execution must continue at the next handler. Otherwise, the catch clause is executed. At the end of the `catch` clause as well as the end of the `try` block, a jump must be added that jumps to after the `try-catch` block or to `finally`. To `finally` an entry label is added.

At the end of each `finally`, the decision must be made to continue traversing upwards to the next handler (a `catch`, `finally`, or end of method), or to exit the `try` and resume regular control flow. This is done by querying the `exc` variable. If it is `null`, normal control flow has to be resumed. If it is not null, control flow should continue at the next handler.

See Rewrite rule 10.

$\underline{\textit{true}}$

```
try {
  ...
} catch (E e) {
  ...
} finally {
  ...
}
```

$\Longrightarrow$

```
try {
  ...
  goto finally_n;
} catch (E e) {
  catch_E:
  if (!(exc instanceof E)) {
    goto finally_n;
  }
  e = exc;  exc = null;
  ...
  goto finally_n;
} finally {
  finally_n:
  ...
  if (exc == null) {
    goto after_try;
  } else {
    goto next_handler;
  }
}
after_try:
```

Rewrite rule 10: Encode `catch`, `finally` and `try` in `goto`.

Encode `throw` Each throw is replaced by an assignment into an "exception variable" named `exc`, and a jump to the next handler. This handler can either be a `catch` clause, a `finally` clause or the end of the method if there is no handler. In the case that an exception can exit the method (i.e. there is no handler), the exception variable must be added as an out parameter, such that it is returned to the caller. Otherwise it can be a local variable. See Rewrite rule 11.

$\underline{\textit{true}}$

```
... throw e; ...
```

$\Longrightarrow$

```
...
exc = e;
goto next_handler;
...
```

Rewrite rule 11: Encode `throw` as `goto`. `next_handler` can either be a label to a catch block or the end of the method.

Encode throwing method calls After each throwing method call the exceptional return value should be checked. If the exceptional return value is non-null, the code should jump to the next handler or end of method. See Rewrite rule 12.

true

```
... x = o.m(exc); ...
```

$\Longrightarrow$

```
...
x = o.m(exc);
if (exc != null) {
  goto next_handler;
}
...
```

Rewrite rule 12: Encode a throwing method call as `goto`. The `exc` out parameter is already present. This was added in the step “Encode throw”.

Inline `try-catch-finally` All `try-catch-finally` components can now also be inlined, since all the control flow is encoded in `goto`. See Rewrite rule 13.

true

```
try {
  ...
} catch (E e) {
  ...
} finally {
  ...
}
```

$\Longrightarrow$

```
...
...
...
```

Rewrite rule 13: Inline `try-catch-finally`.

7.2.5 Encoding `signals`

At this point all control flow is encoded in `goto`s. The only task remaining is to encode signals clauses into regular contracts that reason over the added `exc` output parameter.

Add constraining `signals` A `signals` clause must be added that constrains the dynamic type of `exc`. This is to convey the knowledge that if `exc` is not `null`, the type is constrained by the `signals` clauses of the method. The possible types are derived from all the `signals` clauses. See Rewrite rule 14.

$$\frac{\textit{true}}{}$$

```
//@ signals (E1 e) b1;
...
//@ signals (En e) bn;
void m();
```

$\Longrightarrow$

```
//@ signals (E1 e) b1;
...
//@ signals (En e) bn;
/*@ signals (Throwable t)
      t instanceof E1
      || ...
      || t instanceof En;
  @*/
void m();
```

Rewrite rule 14: Add a `signals` clause that constrains the type of the thrown exception.

Encode `signals` as `ensures` Each `signals` clause only applies when an exception is thrown and the dynamic type of the exception matches the type of the `signals` clause. Each `ensures` clause only applies if no exception is thrown. These two clauses that sometimes apply can be encoded in `ensures` clauses that always apply by making them conditional on the thrown exception. This way, the semantics of `signals` and `ensures` in a language with exceptions can be encoded in `ensures` in a language without exceptions. See Rewrite rule 15.

$$\frac{\textit{true}}{}$$

```
//@ ensures b1
//@ signals (E1 e) b2;
void m();
```

$\Longrightarrow$

```
/*@ ensures
    exc == null ==> b1; @*/
/*@ ensures
    (exc != null
        && exc instanceof E1)
    ==> b2; @*/
void m();
```

Rewrite rule 15: Convert `signals` to `ensures`.

7.3 Correctness

This work does not contain a formal proof of soundness of the suggested transformations. However, three arguments can be made for correctness of the overall approach.

First, the compilation of abrupt termination primitives into exceptions is straightforward. They are conceptually easy to explain, and the implementation is also short and straightforward.

Second, because of this first compilation step, compilation of exceptions can be stated in terms of `throw`, `catch` and `finally`. This simplifies the process in general as there are less edge cases, which means it is less likely that bugs are introduced. This has the added benefit that the transformation of `finally` into `goto` can be stated in terms of the next `finally` and the current `try-catch` block, instead of having to enumerate all loops that wrap this `finally` block and their labels.

Third, each of the transformation steps is modular in the sense that they perform a simple task: compile `continue` into `break`, abrupt termination into exceptions, and exceptions into `goto`, and so on. This aids debugging: correct functioning of each transformation can be checked independently of the others, and writing targeted tests is easier.

7.4 Implementation

This section discusses the implementation of the transformation described in this chapter, and the difference compared to the transformation described in this chapter. The source of the prototype is located on GitHub [59].

The prototype implementation does not completely implement the transformation described in this chapter. Particularly, type checking Java exceptions is not yet included. However, the core of the idea, an ergonomic approach to provide support for abrupt termination, has been implemented.

The passes added to the repository in [59] are:

- `specify-implicit-labels`: Specifies implicit labels, as discussed in Section 7.2.1.
- `continue-to-break`: Encodes `continue` as `break`, as discussed in Section 7.2.1.
- `break-return-to-goto`: Encodes abrupt termination in `goto`, as discussed in Section 7.2.2.
- `break-return-to-exceptions`: Encodes abrupt termination in exceptions, as discussed in Section 7.2.3.
- `intro-exc-var`: This transformation is responsible for managing the `exc` var. This includes possibly adding it as an out parameter, and setting the `catch` variables to the value of `exc`.
- `encode-try-throw-signals`: This is the biggest transformation of the five at 413 lines of code, including comments. It takes some time to read, but the logic is straightforward: each throw is turned into a jump to the next handler. If a `catch` does not handle an exception, it jumps to the next

handler. At the end of a `finally`, a jump is emitted that jumps to the next handler if `exc != null`, which means an exception is being thrown. The next handler is kept track of using push/pop operations as the AST is traversed in a depth first approach.

7.4.1 Differences with theory

There is a difference between the approach described in this work and the implementation. In this work, it is recommended to first encode exceptional control flow into `goto`, and then encode `signals` in `ensures`. In the implementation, there is a separate pass for introducing the `exc` variable, after which a pass is executed that encodes exceptional control flow and `signals` in one go. We did this because the exceptional control flow pass is easier to implement if it can be assumed that the `exc` var already exists. Then, encoding `signals` into `ensures` only took 60 more lines of code, so we decided not to put it into a separate pass.

7.5 Evaluation

This section evaluates if the approach correctly encodes Java semantics. This is done by inspecting the intermediate stages of verification of two examples. While not as rigid as a formal proof, it gives an intuition for correctness of the approach.

The two examples are both from the KeY example set: the KeY abrupt termination challenge example, and an example that shows the interplay between exceptions and abrupt termination [1].

The intermediate stages are acquired by verifying the programs with VerCors, and supplying several flags such as `--show-after` ***`passname`***, where ***`passname`*** is equal to one of the passes named in Section 7.4. Given the `--show-after`, VerCors will output the AST after the given pass has been applied. This is useful for inspecting changes to the AST over time as the verification process progresses.

As the aim of these examples is to highlight points from the proposed transformation, the examples are simplified versions of VerCors output. Specifically, extensively flattened expressions have been recombined, and most contracts and assertions have been removed. However, in the implementation in VerCors, the actual AST has more statements, and annotations and contracts are retained or refined between passes.

7.5.1 Abrupt termination

The following example can be found in our fork of the VerCors repository [61]. We focus on the following `for` loop, as it contains the `break` statement:

```
for (int i = 0 ; i < ia.length; i++) {
  if (ia[i] < 0) {
    ia[i] = -ia[i];
    break;
  }
}
```

The `for` loop iterates over all indices. When a negative element is encountered, the value is flipped, and the for loop is abruptly terminated. First, the standardizing pass is done:

```
__loop_0: for(int i = 0; i < ia.length; i++) {
  if (ia[i] < 0) {
    ia[i] = -ia[i];
    break __loop_0;
  }
}
```

In this case, this pass makes the implicit labels explicit. Then, the pass of encoding `break` to `throw` is done:

```
__loop_0: try{
  for(int i=0;i < ia.length;i ++) {
    if (ia [ i ] < 0) {
      ia [ i ] = - ia [ i ];
      throw new __break___loop_0_ex<>();
    }
  }
} catch (__break___loop_0_ex __ucv_0) { }
```

Normally the `goto` encoding would be used, as there is no `finally` in this program. However, for illustrative purposes, we have forced usage of the `throw` encoding with the `--at-via-exceptions` flag. The `break` is replaced with a `throw`, and the `for` loop is wrapped in a `try-catch` statement.

Finally, exceptional control flow is encoded in `goto`:

```
{
  int i;
  i=0;
  while(i < \length(getOption(this.field_AbruptTermination_ia))) {
    if (getOption(this.field_AbruptTermination_ia)[i] < 0) {
      getOption(this.field_AbruptTermination_ia)[i].item
            = -getOption(this.field_AbruptTermination_ia)[i].item;
      sys__exc = __break___loop_0_ex<>.
          constructor___break___loop_0_ex(current_thread_id,globals);
      goto catch___break___loop_0_ex_2;
    }
    i=i + 1;
  }
  goto try_end_3;
}
{
  label catch___break___loop_0_ex_2;
  if (!TYPE<>.instanceof(\typeof(sys__exc), TYPE<>.
      class___break___loop_0_ex())) {
    goto method_end_internal_AbruptTermination_negatefirst_1;
  }
  __break___loop_0_ex<> __ucv_0;
  assume __ucv_0 == sys__exc;
  sys__exc=null;
  goto try_end_3;
}
label try_end_3;
```

In this code snippet, the `try-catch` statement has been inlined, and been encoded completely in `goto` statements and labels.

Note that between this code snippet and the previous, the program grew. This is because between this pass and the previous pass, other passes have been applied as well, such as encoding references into `Options` or translating `for` loops into corresponding `while` loops.

7.5.2 Exception & abrupt termination interplay

This example shows the interplay between exceptions and abrupt termination. The following example can be found in our fork of VerCors [58]. In the code, a value is returned in a `catch` clause, which is subsequently overwritten by a `return` in the `finally` clause:

```java
//@ ensures \result == 2;
public int bar() {
  // IllegalArgumentException is a subclass of RuntimeException
  IllegalArgumentException e = new IllegalArgumentException();
  int i = 0;
  try {
    throw e;
  } catch (RuntimeException e1) {
    i = 10;
    return 1;
  } finally {
    //@ assert i == 10; // Went through the RunTimeException first
    return 2;
  }
}
```

First, the pre-processing phase is done. It does not make any significant changes, and hence we skip its output. Then the `return` to `throw` transformation is done, because the code contains `finally`:

```java
//@ ensures \result == 2;
int bar(){
  try {
    IllegalArgumentException<> e = new IllegalArgumentException<>();
    int i=0;
    try {
      throw e;
    } catch (RuntimeException<> e1) {
      i = 10;
      throw new __return_bar_ex<>(1);
    } finally {
      assert i == 10;
      throw new __return_bar_ex<>(2);
    }
  } catch (__return_bar_ex<> __ucv_2) {
    return __ucv_2.value;
  }
}
```

A `try-catch` wraps the method body, and each `return` is replaced by a `throw` throwing the returned value wrapped in an exception. Then the exceptional control flow is encoded in `goto`:

```
//@ ensures \result == 2;
int bar(){
    java_DOT_lang_DOT_Object<> sys__exc=null;
    java_DOT_lang_DOT_IllegalArgumentException<> e =
        java_DOT_lang_DOT_IllegalArgumentException<>.
        constructor_java_DOT_lang_DOT_IllegalArgumentException();
    int i = 0;
    {
      sys__exc=e;
      goto catch_java_DOT_lang_DOT_RuntimeException_13;
      goto finally_14;
    }
    {
      label catch_java_DOT_lang_DOT_RuntimeException_13;
      if (!TYPE<>.instanceof(\typeof(sys__exc), TYPE<>.
          class_java_DOT_lang_DOT_RuntimeException())) goto finally_14
          ;
      java_DOT_lang_DOT_RuntimeException<> e1;
      assume e1 == sys__exc;
      sys__exc = null;
      i = 10;
      sys_exc = __return_bar_ex<>.constructor___return_bar_ex(1);
      goto finally_14;
    }
    {
      label finally_14;
      assert i == 10;
      sys__exc = __return_bar_ex<>.constructor___return_bar_ex(2);
      goto catch___return_bar_ex_10;
      if (sys__exc != null) goto catch___return_bar_ex_10;
      goto try_end_15;
    }
    label try_end_15;
    goto try_end_11;
    {
      label catch___return_bar_ex_10;
      if (!TYPE<>.instanceof(\typeof(sys__exc), TYPE<>.
          class___return_bar_ex())) goto method_end_bar;
      __return_bar_ex<> __ucv_2;
      assume __ucv_2 == sys__exc;
      sys__exc=null;
      return __ucv_2.field___return_bar_ex_value;
      goto try_end_11;
    }
    label try_end_11;
    label method_end_bar;
}
```

This increases the code size considerably, but the control flow is still sequential and straightforward. To make the control flow easier to follow, we have explicitly highlighted it in the code. Note how the contract `ensures \result == 2;` is satisfied because the `exc` variable is overwritten with a new return value on line 24.

Chapter 8

Inheritance

This chapter discusses design considerations supporting for Java 7 inheritance, as well as the implementation in VerCors. We first motivate why additional modifications to APFs are needed for verification of Java inheritance in practice. Then three possible approaches and their characteristics are discussed. Based on these three approaches, we present an approach for VerCors. We briefly discuss four informal semantics rules of the suggested approach, and then present our transformation. Finally, we evaluate the implementation by manually applying the approach to an example, and discussing the intermediate steps.

8.1 Design considerations

This section discusses the design considerations for the VerCors approach. We outline the main problem of using APFs for inheritance and discuss the trade-offs of the different approaches.

8.1.1 The APF exchange problem

As discussed in Section 4.2.4, APFs allow verification of behavioural subtyping, but they cannot be applied directly to Java.

The general usage of an APF consists of two steps: a predicate family instance is exchanged for a predicate family entry. This entry is then unfolded like a regular abstract predicate. However, exchanging a predicate family instance for a predicate family entry can only take place if the dynamic type is known. Hence, crucial for using an APF is knowing the dynamic type of the receiver: if the dynamic type is not known, the predicate family instance cannot be used. Furthermore, if merely a subtype relation is known, the predicate family instance also cannot be exchanged for an entry, as the exact type is needed.

In Java, dynamic dispatch ensures that a method call is dispatched to the dynamic type of the object. This suggests that at the start of a method the dynamic type of `this` is known. However, this is not the case: Java allows subclasses to call methods and constructors of superclasses through `super`. Therefore, the only fact known at the beginning of the method is the subtype relation. A method is either executed because of dynamic dispatch, or because a subclass called it using `super`, implying `this instanceof CurrentClass`.

The fact that the dynamic type is needed to use an APF, combined with the fact that the exact dynamic type of an object can only be acquired by testing for it, makes APFs as presented in Section 4.2.4 difficult to use in Java. We refer to this problem as the APF exchange problem. To the best of our knowledge, we have not seen other works state this problem explicitly.

There are two ways to resolve the APF exchange problem. The first option is to limit Java semantics and disallow `super` calls. The second is to extend or adjust the details of how APFs work. Since verification of commercial Java programs is the goal, the second option is explored in this work.

8.1.2 Characteristics of approaches

Adjusting how APFs work is not straightforward: each change has different tradeoffs We have found four characteristics that are affected by different APF approaches: modularity, automatic inheritance of methods, side-calling, and modelling power.

Apart from modularity, each of the characteristics is related to a certain code pattern. Listing 8.1 illustrates how these characteristics can appear concretely in Java programs and specifications. These examples of code patterns are also commented on in this section.

Listing 8.1: Example of automatic inheritance, side-calling and modeling through APF parameters.

```
 1 class Cell {
 2   int value;
 3
 4   // Read only method: should be inherited automatically
 5   int get() {
 6     int res = newVal;
 7     return res;
 8   }
 9
10   // Side-call: reuse implementation of sibling method
11   int setFrom(Cell otherCell) {
12     set(otherCell.get());
13   }
14
15   //@ resource state(int x) = Perm(value, 1) ** value == x;
16
17   // Modeling: model currently held value through APF parameter
18   //@ requires state(oldVal);
19   //@ ensures state(newVal)
20   void set(int newVal) {
21     //@ unfold state(oldVal);
22     this.value = newVal;
23     //@ fold state(newVal);
24   }
25 }
```

Modularity

Modularity indicates whether or not the approach is modular. In the context of inheritance, if an approach is not modular, defining a subclass could cause a parent class to no longer verify, even if it was previously verified as correct. In other words, it means that adding a subclass triggers reverification of the superclass.

Automatic inheritance of methods

Automatic inheritance of methods means that the approach allows inheriting implementations when it is safe to do so, and disallows this if it is not safe to do so. If the approach does not support this, it requires a proof from the user (in the worst case, in the form of extra code) that inheriting the implementation is safe.

An example of this is line 5 in Listing 8.1: the `get` method only reads the program state, and should be inherited automatically without any interaction or extra proofs from the user if it is safe to do so.

Side-calling

Side-calling is when a method calls any of the methods defined in the same class through dynamic dispatch. This is done often in Java code to keep methods small, and reuse code throughout a class.

An example is presented on line 11 in Listing 8.1: the `setFrom` method delegates setting the value to `set`.

Modelling power

Modelling power refers to using APF parameters for modelling the state of an object in an abstract manner. If an APF approach restricts usage of the object when APF parameters are used for modelling state, then the approach does not have full modelling power.

An example of using APF parameters to model object state is shown in Listing 8.1 on lines 15 and 18, where the parameters of the `state` predicate models the value currently held by the `Cell` class. Using the parameter, the contract can specify that a value is set, without specifying how exactly the value is stored internally.

8.1.3 Candidate approaches

From literature and the state-of-the-art tool review we have collected three approaches to make the APF problem manageable. These are the "extension" approach, the "static/dynamic" approach and the "non-modular" approach. Each is discussed next. We have summarised the influence of these three approaches on the four earlier discussed characteristics in Table 8.1.

Approach	Modular	Inherit	Side-calling	Modelling	Used by
Extension	Yes	Automatically	Yes	Limited	Hurlin [36]
Static/dynamic	Yes	Proof needed	No	Full	Verifast [65]
Non-modular	No	Automatically	Yes	Full	Nagini [23]

Table 8.1: This table summarises the three discussed approaches and their characteristics. The "Inherit" column specifies whether or not method implementations can be inherited without user interaction. The "Used by" column indicates examples of the approaches in practice. Note that Verifast and Nagini are also discussed in both Chapters 6 and 9.

Extension

The extension approach was first suggested by Parkinson in *Local reasoning for Java* [54], and later actually used and formalized by Haack and Hurlin in 'Separation

Logic Contracts for a Java-Like Language with Fork/Join' [26], as well as 'Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic' [36].

It solves the APF exchange problem by restricting the definition of APFs: when an APF is defined, it automatically extends the parent APF. This means that unfolding an APF entry results in the body of the APF, and additionally an APF entry of the parent type.

A practical example of this is shown in Listing 8.2 on lines 23 to 28. Here, the dynamic type of `rc` is known, and we assume the constructor returns an APF instance. Therefore, the APF instance can be unfolded. The instance is unfolded into the entry, and then the entry is unfolded as well. This yields a write permission for the field `bak`, as well as another predicate entry `state@Cell`, the superclass of `ReCell`.

Including the APF entry of the superclass by default fixes the APF exchange problem, because it ensures it is always possible to "extract" a predicate entry from a predicate instance. Formally, the hypothetical `extract` statement allows extracting 1) the entry of a given type and 2) a magic wand from an APF instance. The magic wand can be applied to regain the APF instance. The Hoare rule for `extract` is as follows:

$$\frac{o\ \texttt{instanceof}\ C}{\begin{array}{c}\{\ o.p(\bar{e})\ \} \\ \texttt{extract}\ o.p@C(\bar{e}) \\ \{\ o.p@C(\bar{e}) * (o.p@C(\bar{e}) \mathrel{-\!\!*} o.p(\bar{e}))\ \}\end{array}}\ \text{ExtractAPF}$$

The rule states that to extract the predicate entry $o.p@C(\bar{e})$ from $o.p(\bar{e})$, it must be proven that o `instanceof` C. Given that the static type of o always gives a lower bound for C, the `extract` statement is always usable. The validity of the `extract` statement relies on the (Dynamic type) and (`ispartof` Monotonic) axioms from [36].

In short, this approach allows to use an APF, even when only the subtype of the dynamic type is known.

This approach has the drawback that the modelling power is limited. Specifically, when parameters are used in this approach, object state becomes read only. This is shown practically in the `set` method in Listing 8.2. On line 9, a magic wand is received from the extract statement for the value `_`. However, on line 11, the variable that controls this parameter is set to `newValue`. Therefore, to apply the magic wand, a magic wand with a parameter equal to `newValue` is needed, as shown in line 14. However, the only wand available has a parameter equal to `_`, which is not equal to `newValue`, and hence cannot be applied. This example shows that changing the state after acquiring a magic wand through `extract` might render

Listing 8.2: Example of automatic inheritance, side-calling and modeling through APF parameters.

```
class Cell {
  //@ resource state(int x) = Perm(value, 1\1) ** value = x;
  int value;

  //@ requires state(_);
  //@ ensures state(newValue);
  void set(int newValue) {
    //@ extract state@Cell(_);
    //@ assert state@Cell(_) ** (state@Cell(_) -* state(_))
    //@ unfold state@Cell(_);
    value = newValue;
    //@ fold state@Cell(newValue);
    // Does not match the magic wand: apply fails
    //@ apply state@Cell(newValue) -* state(newValue);
  }
}

class ReCell extends Cell {
  //@ resource state(int x) = Perm(bak, 1\1) ** bak = x;
  int bak;
}

// APF contains parent APF:
ReCell rc = new ReCell();
//@ assert rc.state() ** rc.getClass() == ReCell.class;
//@ unfold rc.state();
//@ unfold rc.state@ReCell();
//@ assert Perm(rc.bak, 1\1) ** rc.state@Cell()

// APF can be taken apart ("extracted"):
ReCell rc = new ReCell();
//@ assert rc.state() ** rc.getClass() == ReCell.class;
//@ extract rc.state@Cell();
//@ assert rc.state@Cell() ** (rc.state@Cell() -* rc.state());
```

the wand unusable.

We believe this is an important drawback, but it has not yet been reported as such. We think this is the case because, in ‘Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic’, APF parameters were used for permission amounts [36]. Since permission amounts rarely change, limited modelling power was not a practical problem. However, it might be problematic for other applications of this technique where parameters are used for other purposes.

Static/dynamic

The static/dynamic approach was introduced by Parkinson and Bierman in ‘Separation Logic, Abstraction and Inheritance’ [55] and further refined in ‘Separation Logic for Object-Oriented Programming’ [53].

The static/dynamic approach defines two kinds of contracts: a dynamic contract, which is used when a method is dynamically dispatched, and a static contract, which is used when a method is called through `super` (i.e. statically dispatched).

The contracts have different uses. When a new method is defined there are two proof obligations:

1. The dynamic contract must imply the static contract, given the dynamic type is known.

2. The implementation adheres to the static contract.

These two facts assure that a method satisfies its contracts both during dynamic and static dispatch. An example of a newly defined method is shown in Listing 8.3 on line 6. We will give a short proof of its correctness. Given that the dynamic type is equal to Cell, the dynamic contract implies the static contract, as `state()` can be unfolded into `state@Cell()`, and back into a `state()` after the method. The implementation adheres to the static contract, as it folds a `state@Cell` at the end.

It often happens that the dynamic and static contract are similar. This is the case in Listing 8.3. This similarity can be abbreviated by only specifying the dynamic contract. The static contract is then derived by replacing every occurrence of a predicate family instance with a predicate family entry. In Listing 8.3, the explicit form on line 6 is identical to the implicit form on line 19.

If a method is overriding another, the same proof obligations hold. Additionally, it must be proven that the overriding dynamic contract implies the overridden contract. Specifically, the super-pre-condition must imply the sub-pre-condition. The sub-post-condition must imply the super-post-condition. This is to ensure that

Listing 8.3: Example showing a newly defined method `set` for the `Cell` class.

```
class Cell {
  //@ resource state(int x) = Perm(value, 1\1) ** value == x;
  int value;

  // Explicit form
  //@ dynamic:
  //@   requires state(_);
  //@   ensures state(newValue);
  //@ static:
  //@   requires state@Cell(_);
  //@   ensures state@Cell(newValue);
  void set(int newValue) {
    //@ unfold state@Cell(_);
    value = newValue;
    //@ fold state@Cell(newValue);
  }

  // Implicit form
  //@ requires state(_);
  //@ ensures state(newValue);
  void set(int newValue);

  // Demonstrate impossible side-call
  //@ requires state(_);
  //@ ensures state(val * 2);
  void setDouble(int val) {
    // set is a dynamic dispatch call, hence requires dynamic
       contract
    // Dynamic contract requires state(_)
    // But have state@Cell(_) because of static contract
    // Hence, side-calling is impossible
    set(val * 2);
  }
}
```

Listing 8.4: Example showing a method overriding `set`

```
class ReCell extends Cell {
  /*@ resource state(int x, int y) = Perm(bak, 1\1) ** bak == y
        ** state@Cell(x); @*/
  int bak;

  //@ requires state(oldValue, _);
  //@ ensures state(newValue, oldValue);
  void set(int newValue) {
    //@ unfold state@ReCell(oldValue, _);
    bak = value;
    super.set(newValue);
    //@ fold state@ReCell(newValue, oldValue);
  }
}
```

the Liskov Substitution Principle holds and the sub-method can be used in place of the super-method.

An example of an overriding method is shown in Listing 8.4. In this case, since the dynamic contract is identical to that of the superclass, the super-pre-condition trivially implies the sub-pre-condition. Conversely, the sub-post-condition trivially implies the super-post-condition.

To summarize, this method tries to resolve the APF exchange problem by *always* exchanging the APF instance for an APF entry when a method is dynamically dispatched. This is possible because dynamic dispatch guarantees that the dynamic type is known.

This has one important benefit: it allows super-methods to change the state present in the APF. The static/dynamic contract hierarchy then ensures all predicate entries are packaged back up into a predicate instance again.

However, there are three drawbacks. First, it is a complicated approach, both to explain and use.

Second, side-calling is not possible. If the `set` method from Listing 8.3 needs to call `get`, this is not possible because this would require a predicate instance, while the method implementation of `set` only has access to a predicate entry. This is demonstrated in the method `setDouble` starting on line 23 in Listing 8.3. The method `setDouble` can never call `set`, as it cannot satisfy the dynamic contract of `set`. This shortcoming was first mentioned in [55, Section 5.5], however in our opinion it was addressed too briefly. It is possible to call parent methods through `super`, as these require the static contract, but this is still limiting.

Third, all methods must be overridden. This is because the method might return an abstract predicate family where one of the arguments changed. If so, Listing 8.5: Example showing what happens when a non-modular APF is unfolded. ==> is the implies operator.

```
class C {
  //@ resource p() = body_C;
}
class D extends C {
  //@ resource p() = body_D;
}

C c = ...; // Acquire instance of C or D
//@ unfold c.p();
//@ assert c instanceof C ==> body_C && c instanceof D ==> body_D;
```

the sub-method must then prove that it is safe to wrap this entry into its own entry, which might involve proof steps and variable assignments. For a detailed example we refer the reader to Appendix E.

Non-modular

The non-modular approach is used by the Nagini verifier [23]. While it is used in practice, to the best of our knowledge the details of this approach have not been discussed in any work.

The non-modular approach changes APFs not to have entries, but only instances. These instances can always be unfolded, regardless of the dynamic type of the receiver. When such a non-modular predicate instance is unfolded, it yields the bodies of all classes that define that predicate in the inheritance hierarchy. However, each of these bodies is only included if the current dynamic type is a subtype of the type where that APF is defined. In other words, the body of the instance is dependant on the exact type of the object. If the type is C, only the APF of C is included, if it is D, the APFs of both C and D are included, and so on. Listing 8.5 gives an intuition of what a non-modular APF unfolds to.

There are two benefits to this approach. First, it is straightforward to explain. Second, it allows automatic inheritance of methods. This is because APF bodies of subclasses are included during verification of methods of superclasses. Hence, if a super-method verifies as correct, it can also be used by subclasses.

However, this second benefit is also the drawback: adding a subclass may cause a superclass to no longer verify. In other words, adding a subclass triggers reverification of a superclass with a new APF, which is not modular.

8.2 Chosen approach

This section motivates and describes our suggested approach for supporting inheritance in VerCors. First, we give an overview of the suggested approach, and discuss the main motivations. Then we show and discuss the syntax needed to verify programs with inheritance in VerCors. Then we list the semantic checks that need to be implemented in VerCors to soundly support the suggested approach. Finally, we discuss two proof rules of the informal semantics of the chosen approach included in Appendix D.

8.2.1 Overview & motivation

For VerCors, we recommend to combine the static/dynamic and extension approach. There are two motivations for this recommendation.

First, one of the aims of VerCors is proving functional correctness of concurrent programs. Therefore it is useful to be able to use APF parameters fully, while still being able to mutate the state of objects. The static/dynamic approach allows this while retaining modularity. An example of this is given in Listing 8.6. Because of the static/dynamic approach, the parameters of `state` can be used to model the internal state of `Cell` and `ReCell`, without leaking implementation details. Additionally, the state can be updated without problem.

Listing 8.6: Example showing practical usage of side calling and modelling through APF parameters.

```
 1  class Cell {
 2    int val;
 3    //@ resource state(int x) = Perm(val, 1\1) ** val == x;
 4
 5    //@ requires state(_);
 6    //@ ensures state(newVal);
 7    void set(int newVal);
 8
 9    //@ requires state(currentVal);
10    //@ ensures state(currentVal) ** \result == currentVal;
11    int get();
12  }
13
14  class ReCell extends Cell {
15    int bak;
16    //@ resource state(int x, int y) = Perm(bak, 1\1) ** bak == y;
17
18    //@ requires state(oldVal, _);
19    //@ ensures state(newVal, oldVal);
20    void set(int newVal) {
21      //@ unfold state@ReCell(oldVal, _);
22      bak = super.get(); // Static contracts are used here
23      super.set(newVal);
24      //@ fold state@ReCell(newVal, oldVal);
25    }
26  }
```

Second, the extension approach provides several nice benefits, while not limiting the patterns described in 'Separation Logic for Object-Oriented Programming' [53]. It guarantees side-calling read-only parent methods through `super`, as it ensures a predicate entry is always extractable from a predicate instance. An example can be seen in Listing 8.6, where on line 22 `get` is called throught the parent class `Cell`. Because of the extension approach, `state@ReCell` includes `state@Cell`, which allows `super.get` and `super.set` to be called. The extension approach also integrates well with lock invariants in VerCors. Lock invariants are implemented as abstract predicates with zero arguments, and are therefore well suited for use with the hypothetical `extract` statement, as mentioned in Section 8.1.3.

An example of this is shown in Listing 8.7. Even though the precise dynamic type of `c` is unknown, the extension approach allows us to unpack the `lock_invariant` without knowing the precise dynamic type through the `extract` statement. Normally, after changing state, putting back together the `lock_invariant` would be problematic. However, since the `lock_invariant` does not have any parameters, we can still apply the magic wand to regain the `lock_invariant` APF.

Listing 8.7: Usage of `extract` for manipulation of `lock_invariant`.

```
class Cell {
  int val;
  //@ resource lock_invariant() = Perm(val, 1\1);
}

void doWork(Cell c) {
  synchronized (c) {
    //@ assert c.lock_invariant();
    //@ extract c.lock_invariant@Cell();
    //@ unfold c.lock_invariant@Cell();
    c.val = c.val + 2;
    //@ fold c.lock_invariant@Cell();
    //@ apply c.lock_invariant@Cell() -* c.lock_invariant();
    //@ assert c.lock_invariant();
  }
}
```

The drawback of the suggested approach is identical to that of static/dynamic contracts: automatic inheritance of methods is difficult, and side-calling is not possible. This might be problematic, but more experience with this approach is needed to know this for sure.

8.2.2 New syntax

Several syntactical elements are needed to support usage of the suggested approach. Note that, except for the changes mentioned in the last paragraph about Java syntax elements, all proposed statements and syntaxes are intended for specification. They are used in either ghost code or method contracts, and do not influence the runtime behaviour of the program.

APF entries

```
//@ assert c.state@Cell(x);
//@ assert n.colored@RedNode(y);
```

For expressing assertions about APF entries, the `@` syntax is needed. This allows to qualify an APF with a certain type by adding an `@` followed by the class name after the APF name.

Exchanging instances for entries

```
//@ unfold c.state(x) at Cell;
//@ assert c.state@Cell(x);
//@ fold c.state(x) at Cell;
```

A statement is needed that allows exchanging APF instances for APF entries when the dynamic type is precisely known. We propose extending the `fold` statement to allow specifying a class, in the form of `unfold o.state() at C`. This indicates that `o.state()` should be exchanged for `o.state@C()`. Conversely, this syntax should also be added for `fold`, to allow exchanging an APF entry with an APF instance as well. Using `fold` and `unfold` without `at` should remain possible to fold and unfold plain APs.

Hypothetical `extract` statement

```
//@ assert c.state(x) ** c instanceof Cell;
//@ extract c.state@Cell(x);
//@ assert c.state@Cell(x) ** (c.state@Cell(x) -* c.state(x));
```

Syntax for the hypothetical `extract` statement, as discussed in Section 8.1.3, needs to be added. We have also considered to add a `merge` statement to reverse the effect of the `extract` statement. However, as the effect is also exactly modelled by a magic wand, we have decided not to do this. This is a practical decision: if readability ever becomes a problem, such a shorthand can easily be added.

Changing APF instance arity

```
//@ assert c.state(x);
//@ narrow c.state(x);
//@ assert c.state();
//@ given (int y) widen c.state();
//@ assert c.state(y);
```

To allow for changing arity of APF instances, two statements need to be added: a statement to add a fresh parameter, and another to remove the last parameter. We suggest to implement this via the `widen` and `narrow` statements. The `widen` statement allows to add a parameter to the end of the parameter list of an APF instance. Its syntax is as follows: `given (T x) widen o.p()`, which removes `o.p()` from the current state and adds `o.p(x)`, where `x` is type T. The `narrow` statement removes the last parameter, and only requires the APF instance to be named: `narrow o.p(x)` removes `o.p(x)` from the state and adds `o.p()`.

Contract implication proofs

```
//@ requires P;
//@ ensures Q;
/*@ with {
  // proof steps to prove:
  // super pre-condition implies sub pre-condition
} @*/
/*@ then {
  // proof steps to prove:
  // sub post-condition implies super post-condition
} @*/
void doWork(Cell c) { ... }
```

In some situations proving that a sub-method contract is compatible with the super-method contract is complex. For example, an APF instance might need to be widened with several parameters. At the time of writing this cannot be done automatically. Therefore, separate syntax is needed to include these proof steps for a the method definition. We propose to reuse the `with-then` syntax from VerCors at the method level. `with-then` is already used in VerCors for including proof steps for method calls, and therefore is a logical candidate for proof steps for a method as well. `with-then` should only be allowed for overriding methods.

(Alternative syntax)

```
/*@ implication_proof void doWork(Cell c) {
  // proof steps to prove:
  // super pre-condition implies sub pre-condition
  sub.doWork(c);
  // proof steps to prove:
  // sub post-condition implies super post-condition
} @*/
```

Including the proof steps with the contract of a method definition is a practical decision. Another possibility would be to have the proof as a separate ghost method with a specific prefix or naming. This is the alternative syntax shown above. The method contains several proof steps, then a call to the child method, and then several more proof steps. This mirrors the `with-then` structure, but is a separate syntactical entity from the contract. This might be a useful syntax if implication proofs become long.

Java syntax elements

The Java syntax elements related to inheritance should be supported. This means `extends`, `instanceof`, `super` and casting syntax should be added. These are already supported by the Java parser of VerCors, but additional AST changes might be needed to properly process them.

8.2.3 Semantics to implement in VerCors

Several semantical checks and features need to be implemented in VerCors to fully support the suggested approach and Java inheritance in general. In this section we discuss which checks and semantics are needed, and why.

These semantical checks were determined by inspection of *The Java language specification* [41]. While there might be aspects we have missed, we believe that the checks listed in this section are sufficient for basic support.

Type information

As the Java type system is more complex than the Silver type system, type information will have to be encoded explicitly. Care will have to be taken to ensure type information about arguments, return values, and fields is propagated properly and without user interaction.

Furthermore, Java also enforces type safety by doing runtime type checks. One example of this is unsafe casting, as presented in Listing 8.8. In the example, the type system guarantees `p instanceof Parent`. However, although `Child` extends `Parent`, not every `Parent` is a `Child`. Therefore, casting a `Parent` to a `Child` is an unsafe operation that might fail, unless it is first proven that `p.getClass() == Child.class`. The encoding of type information in COL will have to ensure these checks are done as well. The same goes for emitting proof obligations for preventing `ArrayStoreException`.

Listing 8.8: Example of unsafe casting. It is assumed that `Child` extends `Parent`.

```
Parent p = ...;
Child c = (Child) p; // Unsafe: might throw
```

Distinguish shadowed variables

As described in *The Java language specification* [41], Java allows fields to be shadowed. This means that a subclass can reuse the name of a field of the superclass, which hides the previous field for the subclass. An example of this is given in Listing 8.9, where the `x` field is shadowed in the `Sub` class. Care will have to be taken in the VerCors symbol tables that variable names are correctly scoped, shadowed and hidden.

Listing 8.9: Example of field shadowing. The arrows indicate which definition `x` is referring to. Permissions are elided for brevity.

```
class Super {
  int x;

  //@ ensures x > 0;
  void bound();
}
class Sub extends Super {
  int x;

  //@ ensures x > 10 && Super.this.x > 0;
  void bound();
}
```

Overridden contract compatibility

Whenever methods are overridden and new contracts are specified, compatibility of the super- and sub-contract must be checked. If proofs steps are supplied by the overriding method, as mentioned in Section 8.2.2, these can be verified. If there are no user-supplied proof steps, it must be checked if the compatibility is trivial (i.e. the contracts are identical, or can be checked using boolean implication).

Additionally, as mentioned in Section 8.1.3, it is required to override a method if resources are used in the contract for the static/dynamic approach. This check needs to be implemented. Additionally, an exception can be implemented that allows methods that have a boolean contract to be inherited safely.

Static/dynamic dispatch distinction

Whenever a method is called, the proper contract must be used based on static and dynamic dispatch. For regular method calls, which are dynamically dispatched, the dynamic contract must be used. For static method calls, such as private methods or calls through `super`, the static contract must be used.

Semantics of APFs

Abstract predicate families and the syntaxes proposed in Section 8.2.2 each have their own semantics, as described in Appendix D. These semantics will have to be implemented accordingly and encoded in Silver abstract predicates.

8.2.4 Informal semantics of inheritance rules

Appendix D includes an informal semantics of the constructs needed to support the suggested approach in VerCors. From this list we discuss four rules, as they highlight the complexities of static/dynamic contracts: (FinalCall), (DynamicCall), (NewMethod) and (OverrideMethod).

Method calls

$$\frac{o : C \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ U\ C.m(\bar{X}\ \bar{x})}{\{dynamic(F)\}\ o.m(\bar{e})\ \{dynamic(G)\}}\ \text{(DynamicCall)}$$

$$\frac{o : C \quad C <: D \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public final}\ U\ D.m(\bar{X}\ \bar{x})}{\{static(F)\}\ o.m(\bar{e})\ \{static(G)\}}\ \text{(FinalCall)}$$

The (DynamicCall) rule expresses the semantics of a dynamic method call. This rule is used if the method that is called can be overridden. It contrasts the (FinalCall) rule, which is used for a method call that is `final`, i.e. a method that cannot be overridden. For a dynamic call, the dynamic contract is used. Since the specified contract is by default the dynamic contract, this means nothing is changed. For a `final` call, the static contract is used, i.e. each predicate instance $p(\bar{e})$ is replaced with a predicate entry $p@C(\bar{e})$.

Declaring methods

$$\frac{C\ extends\ D \quad m \notin D \quad \{static(F)\}\ \bar{c}\ \{static(G)\}}{\texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public [final]}\ U\ C.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \}}\ \text{(NewMethod)}$$

$$\frac{\begin{array}{ll} & \texttt{requires}\ F';\ \texttt{ensures}\ G';\ \texttt{public}\ U\ D.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \} \\ \{static(F)\}\ \bar{c}\ \{static(G)\} & dynamic(F') \mathrel{-\!\!*} dynamic(F) \\ C\ extends\ D & dynamic(G) \mathrel{-\!\!*} dynamic(G') \end{array}}{\texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public [final]}\ U\ C.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \}}\ \text{(OverrideMethod)}$$

The (NewMethod) rule expresses the semantics of defining a new method. It requires that the method does not exist in the superclass, and that the implementation adheres to the static contract.

A proof obligation that is missing here is the compatibility between the static and dynamic contract: that the dynamic pre-condition implies the static pre-condition, and the static post-condition implies the dynamic post-condition. However, this proof obligation is not necessary. The contract of the method is assumed to be the dynamic contract, and the static contract is derived from that. Deriving the static contract is proven sound in [53].

The (OverrideMethod) rule applies when the method does exist in the superclass. In this case, it must additionally be proven that the method contract implies the super-method contract. This is indicated by the two separating implications. Proving that the contract implies the super-method contract might be complicated. For example, the proof might involve changing the arity of APFs, and folding and unfolding of abstract predicates. Therefore, in some cases the user might have to supply a proof with the method contract, which VerCors can then verify.

8.3 Transformation of inheritance

To implement the semantic checks outlined in Section 8.2.3, transformations on the AST need to be implemented in VerCors. They are categorised into five phases, and are summarised in Figure 8.1. We give a brief overview of these phases, and then discuss each of the phases and their internal transformation steps in detail.

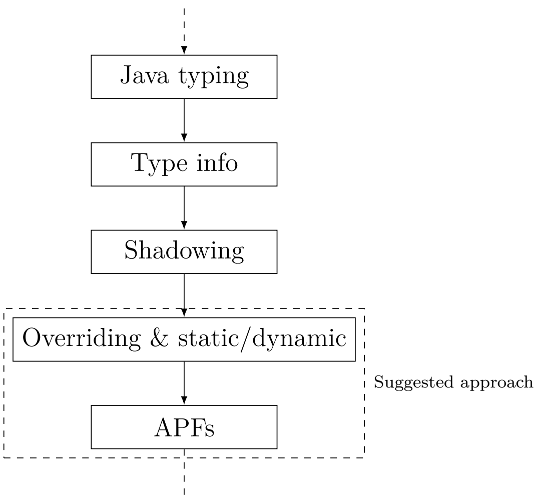


Figure 8.1: Summary of the transformation of a language with inheritance and abstract predicate families to a language without inheritance and only abstract predicates.

The first phase is the Java typing phase, which checks if all typing rules related to inheritance outlined in *The Java language specification* are adhered to. Since this is not the main focus of this work it is not discussed.

The second phase is the type info phase, which contains all typing-related transformations. It encodes the Java type system in COL. This phase is discussed in Section 8.3.1.

The third phase is the shadowing phase. It ensures fields are properly namespaced by class, such that subclasses can reuse names already defined in their superclass. It is discussed in Section 8.3.2.

The fourth phase is the overriding & static/dynamic contract phase. It implements the semantics of static/dynamic contracts as discussed in Section 8.1.3, which ensures the Liskov Substitution Principle is enforced. It is discussed in Section 8.3.3.

Finally, the fifth phase is the APF phase. It ensures APFs are encoded in plain abstract predicates, and it implements the semantics of the new statements suggested in Section 8.2.2. It is discussed in Section 8.3.4.

8.3.1 Typing

The typing phase ensures the Java type system is correctly encoded in COL. It consists of two steps:

Encode type information in COL To encode Java type information in COL, an axiomatic data type (ADT) must be added to the program. This ADT contains all the types used by the program, and expresses the inheritance relations between these types. As ADTs are not the focus of this work, we do not discuss the details of the ADT. For more information about ADTs we refer the reader to [62]. An example of how the ADT can be defined for a small number of types is included in Appendix F. The type ADT in Appendix F was designed and implemented for the Nagini verifier by Marco Eilers [23]. However, because of its generality, we believe it can be used for Java verification in VerCors with few changes.

Once this type ADT is included, the type ADT functions must be added at places in the AST where the type information is known, such that the type information can be used to perform checks that would normally be implemented through run-time checks. Including type information about return values, method arguments and newly created objects is straightforward, as the type information can be added to contracts or included with the `new` call. An example of this is shown in Rewrite rule 16.

true

```
C c = new C();
```

$\Longrightarrow$

```
C c = new C();
//@ assume typeof(c) ==
    class_C();
```

Rewrite rule 16: Add explicit type information at `new`. The `typeof` and `class_C` functions are introduced by the type ADT.

However, for object fields this is more complicated. As field accesses can appear in many places, it is not immediately clear where this type information should be added. To resolve this, we apply the same trick that is used by Brodowsky: we attach type information to permissions [10]. This works because, to use fields, permissions are needed. Therefore, if type information is attached to permissions, whenever a field is used the type information will be present as well. An example of this transformation is shown in Rewrite rule 17.

`f : F`

```
Perm(x.f, 1\2)
```

$\Longrightarrow$

```
Perm(x.f, 1\2) **
  (x.f != null
    ==> x.f instanceof F
      )
```

Rewrite rule 17: Shows how `Perm` can be annotated with type information. It is assumed the static type of `f` is `F`

Introduce proof obligations for runtime type checks When type information is properly encoded in COL, checks can be added that enforce the type system rules. Particularly, we present two checks to prevent `ClassCastException` and `ArrayStoreException` in Rewrite rule 18.

```
Parent p = ...;
Child c = (Child) p;
Parent[] p_arr = ...;
Parent p2 = ...;
p_arr[0] = p2;
```

$\Longrightarrow$

true

```
Parent p = ...;
//@ assert p instanceof Child;
Child c = (Child) p;
Parent[] p_arr = ...;
Parent p2 = ...;
//@ assert p2 instanceof elemtype(
    typeof(p_arr));
p_arr[0] = p2;
```

Rewrite rule 18: Insert type checks when casting. The `elemtype` function returns the type of the elements of a given array type.

In our opinion, the second `assert` in the above rewrite rule models a complex behavior of the Java type system: the dynamic type of `p2` must be a subtype of the dynamic element type of the `p_arr` array. This must be checked because the dynamic type of `p_arr` might be `Child[]`, or an array of any other subtype of `Parent`. Therefore, assigning a `Parent` in a `Parent[]` is not always safe to do, and must be checked.

8.3.2 Shadowing

The shadowing phase ensures that fields can be properly shadowed and field names can be reused, as discussed in Section 8.2.3. This is done in a single step: each class name must be prepended to the field name. This is to ensure that each field name is unique within the inheritance hierarchy, as Silver does not support duplicate field names. An example of this transformation is shown in Rewrite rule 19.

true

```
class A {
  int x;
}
class B extends A {
  int x;
  ...
  x = 3;
  A.this.x = 5;
}
```

$\Longrightarrow$

```
class A {
  int A_x;
}
class B extends A {
  int B_x;
  ...
  B_x = 3;
  A_x = 5;
}
```

Rewrite rule 19: Fields are qualified by class name.

8.3.3 Overriding & static/dynamic

The overriding & static/dynamic phase implements the static/dynamic contract semantics discussed in Section 8.1.3, and ensures methods are inherited and overridden safely. This happens in four steps:

Provide default implementations First, default implementations are provided where methods are not overridden. These default implementations implement the semantics of a non-overridden method: they inherit the contract, and call the method in the superclass. Because shadowed variables have been distinguished in an earlier phase, the contracts do not have to be modified. Rewrite rule 20 shows an example of adding such a default implementation.

Note that providing this default implementation does not mean it is safe to inherit the method. As explained in Section 8.1.3, even a default implementation might need proof steps to inherit a read-only method. Therefore, this approach only allows automated inheritance of methods with trivial contracts: the contracts can only contain permissions and boolean assertions. Contracts using APFs in complex ways will not verify without additional proof steps. In short, if proof steps are necessary for inheriting this method, verification will fail when checking this default implementation.

$$\frac{\textit{true}}{}$$

```
class A {
  //@ requires P;
  //@ ensures Q;
  int m(int x);
}
class B extends A {
}
```

$\Longrightarrow$

```
class A {
  //@ requires P;
  //@ ensures Q;
  int m(int x);
}
class B extends A {
  //@ requires P;
  //@ ensures Q;
  int m(int x) {
    return super.m(x);
  }
}
```

Rewrite rule 20: Provide default implementations.

Check dynamic contracts A proof obligation must be produced that checks if the dynamic contract of a subclass method is compatible with a superclass method. This enforces the Liskov Substitution Principle, such that sub-methods can be used in place of super-methods. If proof steps are supplied with the method definition to prove this compatibility, they are included here.

The proof obligation is modelled as a new method on the subclass. The method on the subclass has the contract of the method of the superclass, and calls the method of the subclass. This ensures the super-pre-condition implies the sub-pre-condition, and the sub-post-condition implies the super-post-condition. An example of this transformation is shown in Rewrite rule 21. Note that within `compatible_m`, the method `m` is called through dynamic dispatch, which ensures the dynamic contract of `B.m` is used.

$$\frac{\textit{true}}{}$$

```
class A {
  //@ requires P;
  //@ ensures Q;
  void m(int x);
}
class B extends A {
  //@ requires R;
  //@ ensures S;
  //@ with C;
  //@ then D;
  void m(int x);
}
```

$\Longrightarrow$

```
class A {
  //@ requires P;
  //@ ensures Q;
  void m(int x);
}
class B extends A {
  //@ requires R;
  //@ ensures S;
  void m(int x);

  //@ requires P;
  //@ ensures Q;
  void compatible_m(int x) {
    C;
    m();
    D;
  }
}
```

Rewrite rule 21: Check compatibility of dynamic contracts

Check static contract The implementations of methods must be checked against the static contracts. This is also modelled by introducing an additional method where instead of the dynamic contract the static contract is used. To translate the dynamic contract into a static contract, each APF instance with receiver `this` is replaced with an APF entry of the current class. An example of this transformation is shown in Rewrite rule 22.

$$\frac{\textit{true}}{}$$

```
class A {
  //@ requires this.state();
  //@ ensures this.state();
  void m(int x) {
    ...
  }
}
```

$\Longrightarrow$

```
class A {
  //@ requires this.state();
  //@ ensures this.state();
  void m(int x);

  //@ requires this.state@A();
  //@ ensures this.state@A();
  void static_m(int x) {
    ...
  }
}
```

Rewrite rule 22: Check implementation against static contract

Distinguish contract usage When methods are dynamically dispatched, the dynamic contract should be used. When methods are statically dispatched, the static contract should be used. This is enforced by distinguishing between calling the dynamic or static method for each method call. When a method is called through `super`, or a method is private, static dispatch is used. Otherwise, dynamic dispatch is used.

To be able to refer unambiguously to the super-method, duplicate and overridden method names need to be resolved. This is done by prefixing each method with the name of the class it is defined in. This way, each class can refer to methods of the parent unambiguously, similarly to shadowed fields.

Rewrite rule 23 shows an example of how this transformation is applied to method calls. Note that, for the static dispatch call, the static contract method introduced in the previous step is used, to force usage of the static contract of the super-method.

$$\underline{\textit{true}}$$

```
class B extends A {
  ...
  m(3);
  super.m(5);
  ...
}
```

$$\Longrightarrow$$

```
class B extends A {
  ...
  B_m(3);
  A_static_m(5);
  ...
}
```

Rewrite rule 23: Force usage of dynamic contract for dynamic dispatch, and static contract for static dispatch.

8.3.4 APFs

In the last phase, APFs are encoded into plain abstract predicates. This is done through the following steps:

Encode semantics of new syntax The semantics of the new syntaxes discussed in Section 8.2.2 must be encoded in `inhale/exhale` statements. Most of the proposed statements can be encoded in two actions: first removing a certain APF from the state, and then adding a different APF to the state. Combined with the specification of the semantics of the statements in Appendix D, the statements are easily encoded into an `exhale` and `inhale`.

`inhale` and `exhale` are similar to the `assume` statement in regular program logics. `inhale` adds a certain assertion to the current state. `exhale` asserts the assertion, and then removes it from the current state. If `exhale` cannot assert the assertion, the program cannot be verified.

With these two statements, arbitrary permissions and APFs can be added and removed from the program state. However, particularly `inhale` must be used carefully, as it can introduce unsoundness when false facts or resources are inhaled.

For example, on the right side in Rewrite rule 24, `this.p(x)` is first checked to be true. If so, it is removed from the state. Then, `this.p(x, y)` is added to the program state.

While encoding most of the new syntaxes is straightforward, `given-widen` is different from the other new syntaxes, because it introduces a new name into the scope. We encode this in COL by inserting a new variable between the `exhale` and `inhale`. After the `given-widen`, this new variable can be used to make assertions about the APF. During type checking, it is ensured that the variable name used in `given-widen` is unique. Since no assertions can be made about the new introduced variable before the `given-widen` statement, the encoding is safe.

A brief example of this transformation is shown in Rewrite rule 24.

true

```
/*@ given (int y)
    widen this.p(x); @*/
```

$\Longrightarrow$

```
//@ exhale this.p(x);
//@ int y;
//@ inhale this.p(x, y);
```

Rewrite rule 24: Encode `given-widen` into `exhale` and `inhale`.

Encode APFs into plain abstract predicates APFs can now be encoded in plain abstract predicates. For each APF declaration, a plain abstract predicate is created where the types of the arguments are included in the name of the APF. This abstract predicate can never be unfolded, as it needs to be exchanged with an APF entry first. Therefore, this new plain abstract predicate is not given a body: unfolding this plain abstract predicate is not allowed, and is considered a bug. This transformation is shown in Rewrite rule 25.

true

```
// In class A:
/*@ resource state(int x) =
      x == 5; @*/
// In class B:
/*@ resource state(int x, int y) =
      y == 8; @*/
```

$\Longrightarrow$

```
// In class A:
/*@ resource state(int x) =
      x == 5; @*/
// In class B:
/*@ resource state(int x, int y) =
      y == 8; @*/
// Outside both classes:
//@ resource state_int(int x);
/*@ resource
    state_int_int(int x, int y);
  @*/
```

Rewrite rule 25: Encode APF declarations into plain abstract predicates. It is assumed that predicates defined outside any class are plain abstract predicates, which can always be folded and unfolded.

Then, for each APF declaration, an APF entry must be encoded in a plain abstract predicate. This is done by creating a new plain abstract predicate ending in the name of the class. The body of this predicate is identical to the body of the APF declaration, in addition to an APF entry of the superclass. This is needed to adhere to the extension approach, as discussed in Sections 8.1.3 and 8.2, which allows several useful code patterns. This transformation is shown in Rewrite rule 26.

`B extends A`

```
// In class B:
/*@ resource state(int x) =
      x == 5; @*/
```

$\Longrightarrow$

```
// In class B:
/*@ resource state(int x) =
      x == 5; @*/
// Outside any class:
/*@ resource state_B(int x) =
      x == 5 ** state_A(x); @*/
```

Rewrite rule 26: Encode APF declarations into plain abstract predicates.

Finally, each use of an APF instance or APF entry can be replaced by the appropriate version of the previously generated plain abstract predicates. This exchange is shown in Rewrite rule 27.

`B extends A`

```
// Abstract predicate instance:             // Abstract predicate instance:
//@ assert x.state(5);                      //@ assert x.state_int(5);
// Abstract predicate entry:           ⟹    // Abstract predicate entry:
//@ assert y.state@Cell(8);                 //@ assert y.state_Cell(8);
```

Rewrite rule 27: Replace APF usage with plain abstract predicates.

8.4 Evaluation

Because of time constraints, we have not implemented a prototype of the suggested approach in VerCors. However, to showcase the approach, as well as give an intuition for how it works, we apply the phases of the transformation manually to an example program, and discuss the intermediate programs.

The example which is used for the evaluation is, to the best of our knowledge, a correct annotated version of an example that often appeared in this work: the Cell/ReCell example. The example is complete, except for constructors, to make the example fit on one page. We list the example fully in Appendix G. Below, and for the rest of this section, we will only list the `ReCell` class and the `set` method, as the changes to the other methods and the `Cell` class are similar.

```
class ReCell extends Cell {
  int bak;
  //@ resource state(int x, int y) = Perm(bak, 1\1) ** y == bak;

  //@ requires state(oldVal, oldBak);
  //@ ensures state(newVal, oldVal);
  /*@ with {
        given (int oldBak) widen state(oldVal);
      }; @*/
  /*@ then {
        narrow state(newVal, oldval);
      }; @*/
  void set(int newVal) {
    //@ unfold state@ReCell(x, y);
    bak = super.get();
    super.set(newVal);
    //@ fold state@ReCell(x, y);
  }
}
```

First, the typing phase is applied. Changes from this phase are minimal: only the `Perm` expressions are exchanged for `Perms` that include type information, as discussed in Section 8.3.1. Therefore, this intermediate program is not discussed.

Then, the shadowing phase is applied, resulting in the following partial program:

```
 1  class ReCell extends Cell {
 2    int ReCell_bak;
 3
 4    /*@ resource state(int x, int y) =
 5              Perm(ReCell_bak, 1\1) ** y == ReCell_bak; @*/
 6
 7    //@ requires state(oldVal, oldBak);
 8    //@ ensures state(newVal, oldVal);
 9    /*@ with {
10          given (int oldBak) widen state(oldVal);
11        }; @*/
12    /*@ then {
13          narrow state(newVal, oldval);
14        }; @*/
15    void set(int newVal) {
16      //@ unfold state@ReCell(x, y);
17      ReCell_bak = super.get();
18      super.set(newVal);
19      //@ fold state@ReCell(x, y);
20    }
21  }
```

The field names are now unique, as they are prefixed by the name of the class. Note how `bak` was renamed to `ReCell_bak` at both the definition on line 2 and the usage on line 17.

Then, the static/dynamic phase is applied:

```
 1 class ReCell extends Cell {
 2   int ReCell_bak;
 3
 4   /*@ resource state(int x, int y) = Perm(ReCell_bak, 1\1)
 5         ** y == ReCell_bak; @*/
 6
 7   //@ requires state(oldVal, oldBak);
 8   //@ ensures state(newVal, oldVal);
 9   void ReCell_set(int newVal);
10
11   //@ requires state@ReCell(oldVal, oldBak);
12   //@ ensures state@ReCell(newVal, oldVal);
13   void ReCell_static_set(int newVal) {
14     //@ unfold state@ReCell(x, y);
15     ReCell_bak = Cell_static_get();
16     Cell_static_set(newVal);
17     //@ fold state@ReCell(x, y);
18   }
19
20   //@ requires state(oldVal);
21   //@ ensures state(newVal);
22   void compatible_set(int newVal) {
23     //@ given (int oldBak) widen state(oldVal);
24     ReCell_set(newVal);
25     //@ narrow state(newVal, oldval);
26   }
27 }
```

The phase applied three changes to the partial program.

First, the `set` method is split up in two: the dynamic and the static definition. The dynamic definition starting on line 7 models the dynamically dispatched method, which has the dynamic contract of `set`. The dynamically dispatched implementation reuses the statically dispatched implementation, so the dynamically dispatched method has no body.

The static definition starts on line 11, and it models the statically dispatched method. The statically dispatched method also checks if the method implementation adheres to the static contract. Therefore, in contrast to the dynamically dispatched method, it does have a body. Note that the contact of `ReCell_static_set` was also changed: each APF instance in the contract was replaced by the APF entry `state@ReCell`.

Second, ambiguous method names are now distinguished by class name at both the definition and where they are called. Furthermore, each method definition and call now indicates if it uses the static or dynamic contract. For example, on line 16, `Cell.set` was originally called through `super`, which is a statically dispatched call.

Now, the statically dispatched method & contract of `Cell.set` is used.

Third, the contract compatibility of `Cell.set` and `ReCell.set` is checked through the generated method `compatible_set`. In this case, the proof steps supplied in the original program are also used in the compatibility proof.

If the proof steps were not supplied, the body of `compatible_set` would consist of only the call `ReCell_set(newVal)`. This would not be verifiable, as the `state` predicate would not have the correct number of arguments for `ReCell_set`. However, in this case, because of the proof steps that add the `oldBak` argument, the method verifies.

Now that the static/dynamic contract semantics are encoded, the APF phase is applied:

```
//@ resource state_int(Object this, int x);
//@ resource state_int_int(Object this, int x, int y);
/*@ resource state_Cell(Cell this, int x) =
        Perm(Cell_val, 1\1) ** x == Cell_val; @*/
/*@ resource state_ReCell(ReCell this, int x, int y) = state_Cell(x)
        ** Perm(ReCall_bak, 1\1) ** y == ReCell_bak; @*/

class ReCell extends Cell {
  int ReCell_bak;

  //@ requires state_int_int(this, oldVal, oldBak);
  //@ ensures state_int_int(this, newVal, oldVal);
  void ReCell_set(int newVal);

  //@ requires state_ReCell(this, oldVal, oldBak);
  //@ ensures state_ReCell(this, newVal, oldVal);
  void ReCell_static_set(int newVal) {
    //@ unfold state_ReCell(this, x, y);
    ReCell_bak = Cell_static_get();
    Cell_static_set(newVal);
    //@ fold state_ReCell(this, x, y);
  }

  //@ requires state_int(this, oldVal);
  //@ ensures state_int(this, newVal);
  void compatible_set(int newVal) {
    //@ exhale state_int(this, oldVal);
    //@ int oldBak;
    //@ inhale state_int_int(this, oldVal, oldBak);
    ReCell_set(newVal);
    //@ exhale state_int_int(this, newVal, oldVal);
    //@ inhale state_int(this, newVal);
  }
}
```

In the final partial program, three major changes are visible.

First, the `state` APF is split up into four distinct plain abstract predicates on line 1: two relating to the APF instances with no bodies, and two relating to the APF entries with bodies. Each use of an APF instance or entry has also been replaced with the appropriate plain abstract predicate, such as on lines 11 and 18

Second, `given-widen` and `narrow` have been replaced by pairs of `exhale` and `inhale`, reflecting the semantics of the statements.

Third, the receivers of the APF `state` have been lifted into the first argument of the APF `state`. This is needed because plain abstract predicates do not have receivers, but are stand-alone predicates independent of a class.

To conclude, after applying the transformation suggested in this chapter, implicit semantics such as overriding, field shadowing, static/dynamic dispatch, and APF extension have been made explicit. The only aspects of inheritance that remain are using fields from superclasses, and calling methods from superclasses. These two aspects can be handled by the transformation that removes the class hierarchy from COL: `silver-class-reduction`. This transformation is responsible for transforming class methods into free functions, and aggregating all fields in the program in one place. The implementation of `silver-class-reduction` will have to be adjusted, but we believe this will be straightforward as the AST contains only unambiguous references at this point. As `silver-class-reduction` is already part of VerCors, we will not discuss it here.

Chapter 9

Related Work

There are eight practical checkers that support exceptions and/or inheritance with separation logic, each with their own level of support. These are: Nagini, Verifast, jStar, Key, OpenJML, JaVerT, Krakatoa and VerCors. Of these, Nagini, jStar and Verifast have the highest level of support, and are comparable to VerCors. They are discussed first. Then the other five checkers are discussed.

9.1 Nagini

Exceptions Nagini fully supports exceptions in the Python language. This means it supports the Python equivalents of the statements `break`, `continue`, `return`, `try`, `catch`, and `finally`. This is done by encoding the control flow into an auxiliary state variable that indicates if the function is currently breaking, continuing, returning, or throwing. We have concluded this after examination of the source code and by manually inspecting the output of the tool.

At first sight it seems that the Python exception model is identical to the exception model of Java. However, there is one subtle difference: Python does not allow labeled breaks. As labeled `break`s complicate the verification of `finally` (explained in Section 7.1.4), the implementation strategy employed by Nagini is not directly usable for verifying exceptions in Java and would have to be extended.

It is interesting to note that while exploring the capabilities of Nagini regarding exceptions, we discovered a bug in Nagini [60]. It was quickly fixed by the developers after reporting it.

The main difference between VerCors and Nagini with regard to exceptions is that Nagini has a single-pass architecture and VerCors a multi-pass architecture. The auxiliary state approach fits the single-pass architecture, as it only requires the ability to emit queries and mutations of the control-flow variable. It also happens to be efficient, at the expense of extra bookkeeping. However, it is not flexible.

This is not a problem for Nagini as Python does not have a statement similar to labeled `break`. The multi-pass architecture of VerCors allows the more flexible and modular multi-pass approach, at the expense of needing multiple passes. This is necessary to support labeled `break` while at the same time keeping VerCors as maintainable as possible. This is a classic trade-off: the single-pass approach is more complex and performant, while the multi-pass approach is less complex but less performant.

Inheritance Nagini has broad support for inheritance in Python. As long as the input is annotated with types as specified in PEP 484 [57], Nagini can reason about the types and subclasses of objects. For semantics of types, Nagini adopts the model of MyPy [69].

Nagini uses non-modular APFs, as discussed in Section 8.1.3. We have concluded this after inspection of the source code and tool output. In Listing 9.1 an example of inheritance as supported by Nagini is given. Here `Parent` and `Child` both define an entry in the APF `abstract_predicate_family`. Then, at line 29, the predicate instances of `Parent` and `Child` can be used, but only if the dynamic type is a subtype of those types.

There is one technical issue that Nagini worked around in an interesting way. The problem is that of folding its non-modular APFs. See again Listing 9.1, line 9. Here the APF `abstract_predicate_family` is folded because the constructor needs to satisfy the postcondition. This is fine if the constructor runs for `Parent`, but it is problematic if the dynamic type of `self` is `Child`. When the dynamic type is `Child`, *childBody* also needs to hold, which is not the case. Hence this should result in an error.

Nagini works around this by inlining the constructor for the `super` call at line 18. Additionally, when the method is inlined, `Fold` statements are not included. This way *parentBody* stays in the scope for the later `Fold` at line 22. This is illustrated by the pair of `Assert`s at line 19.

This approach is effective as long as the user stays within the intended uses of APFs. Once users start wrapping APFs inside regular predicates, problems can occur. A pathological example of such a situation is given in appendix C.

9.2 Verifast

Exceptions Verifast almost fully supports Java exceptions. This means `break`, `return`, `continue`, `throw`, `try`, and `catch` are all supported. These are encoded directly into SMT. The only language feature missing is `finally`. As mentioned in [38], the authors of Verifast are not yet sure what would be an acceptable way

Listing 9.1: Brief example usage of inheritance and APFs as supported by Nagini.

```
class Parent:
  @Predicate
  def abstract_predicate_family():
    return parentBody

  def __init__(self):
    Ensures(self.abstract_predicate_family())
    ...
    Fold(self.abstract_predicate_family())

class Child(Parent):
  @Predicate
  def abstract_predicate_family():
    return childBody

  def __init__(self):
    Ensures(self.abstract_predicate_family())
    super(Child, self).__init__()
    Assert(parentBody) # Proven
    Assert(self.abstract_predicate_family()) # Rejected
    ...
    Fold(self.abstract_predicate_family())

def foo(p: Parent):
  Requires(p.abstract_predicate_family())
  # Contract ends here

  Unfold(p.abstract_predicate_family())
  Assert((parentBody if isinstance(p, Parent) else True)
      and (childBody if isinstance(p, Child) else True))
```

of encoding `finally` clauses, which is the reason why it does not support it at the time of writing.

Inheritance Verifast has mature support for inheritance, as introduced in 'VeriFast for Java' [65]. It supports modular verification of inheritance using static/dynamic contracts and APFs, as discussed in Section 8.1.3. Verifast APFs follow the optionally extended form, as outlined in [54]. That means that when an APF entry is unfolded, it is not guaranteed a predicate entry of the superclass is included. The static/dynamic contracts of Verifast are automatically derived from one single contract to allow for succinct specification of methods.

Support for inheritance in Verifast is limited in two specific ways.

First, it is not possible to provide static and dynamic contracts separately. While Parkinson and Bierman suggest most patterns can be implemented without separate contracts [53], if this is at some point needed for fine-grained control this could be a problem when using Verifast. Our suggested for VerCors also does not allow this.

Second, Verifast APFs have a fixed number of arguments. This means that if a subclass is added that wants to expose additional state through an extra parameter, the base class will have to be changed. This limitation is manageable if regular refactorings of parent classes are acceptable. Our suggested VerCors does allow a form of variable arity APFs.

Since Verifast uses the static/dynamic approach, all limitations discussed in Section 8.1.3 apply. While we have no concrete discussions or citations confirming this, we suspect the Verifast developers are aware of the side-calling limitation of the static/dynamic approach. As an example of this we refer the reader to one of the examples of the Verifast example directory at [56]. Here, the `final` attribute is needed when declaring the `ArrayList` class. If this attribute is removed, the `addAll` method cannot be verified, as it calls the `add` method.

9.3 jStar

Exceptions From manual inspection of the source, we have concluded that jStar supported exceptions up to `finally`, similar to Verifast. Because `break` and `finally` are not considered, we expect that their support for exceptions has been implemented through `goto` or direct support in their symbolic execution implementation.

Inheritance Similar to VerCors, jStar supports inheritance through APFs and the static/dynamic approach. However, besides that it supports two novel features that we nor Verifast or Nagini supports. These are automatic inheritance of methods, and support for manually specifying both static and dynamic contracts. jStar also supports variable argument APFs, similar to our suggested approach.

Variable argument APFs are supported through width subtyping. Type T is a width subtype of U if T has at least every field that U also has by name. However, T might also have fields that U does not have. An example definition of an APF in jStar is presented in Listing 9.2, paraphrased from [19].

Listing 9.2: Example of defining an APF in jStar

```
class Cell {
  define Val(x, content = y) as  x.<Cell: int val> |-> y
}
```

The APF `Val` is an APF with two arguments. First there is `x`, which is a reference to an object with a `val` field. Second there is the value `y`, which should be equal to the value pointed to by `x.val`. This APF can be extended by a subclass as follows:

Listing 9.3: Example of extending an APF in jStar

```
class ReCell extends Cell {
  define Val(x, content = y; old = z) =
      Val$Cell(x, {content = y}) * x.<Recell: int bak> |-> z
}
```

This extended `Val` APF defines an extra argument, `old`, which indicates the previous value `x.val` had. It also includes the APF entry from `Cell`, as indicated by the `Val$Cell` predicate. Because of width subtyping, these two predicate definitions can both be used. If the `old` argument is not needed, it can be omitted. If it is not present but needed, it is existentially quantified. For example, the `<init>` method of Cell can have the following contract shown in Listing 9.4. In this case, the `content` field is existentially quantified: it has a value, but its specific value is not given.

Listing 9.4: Example of a constructor with a contract existentially quantifying the `content` field. `<init>` is the name of the method in jStar that models the constructor.

```
void <init>() :
  {}                              // Precondition
  { Val$(this, {content=_})}      // Postcondition
```

Automatic inheritance of methods is facilitated through the axiom system in jStar. This axiom system allows users to define axioms, which are assumed to be true by the prover in jStar. Of course each axiom should be accompanied by a hand-written proof. These axioms allow jStar to prove inheritance of methods correct, as long as the set of provided axioms is sound.

jStar allows the user to specify static and dynamic contracts separately. If this is not needed the user can use a shorthand of using a `$` after an APF. This causes jStar to derive the static and dynamic contracts in a way similar to Verifast.

9.4 KeY

KeY supports Java exceptions fully. KeY is based on the JavaDL logic, as described in [1]. JavaDL provides axiomatic rules for dealing with exceptions and labeled breaks. Therefore, to handle Java, `continue` and `return` must be transformed into labeled breaks. Extra flag variables are also added to keep track of what kind of control flow is happening. This is needed such that when a loop terminates, it can be deduced if it was normal termination or some kind of abrupt termination. *Deductive Software Verification - The KeY Book* by Ahrendt et al. [1] describes this transformation.

As described in [68], this approach is soon to be replaced by the loop scopes approach, described in the same paper. This new approach reduces the number of proof obligations KeY generates in most cases when dealing with abrupt termination in loops. This is achieved by encoding the kind of abrupt termination that is happening in the axiomatic rules, instead of putting it in auxiliary state variables as in the previous approach.

9.5 OpenJML

OpenJML also supports the full subset of Java exceptions, as well as extensive JML support for specifying the behaviour of exceptions. In [68] it is mentioned that exceptions and abrupt termination are implemented in OpenJML by encoding the control flow in `goto`.

9.6 JaVerT

JaVerT supports exceptions as defined in ECMAScript 5 Strict mode fully. Unfortunately we have not been able to get JaVerT to compile and run. However, from the implementation we have deduced that JaVerT takes the approach of compiling exceptions directly to gotos. It is currently unclear to us if JaVerT handles `finally` correctly, as it cannot be clearly discerned from the implementation if `finally` blocks are inlined, or if auxiliary variables are used to keep track of control flow.

9.7 Krakatoa

Krakatoa supports exceptions, but the level of support is unclear. It accomplishes this by compiling Java exceptions into the more limited exception model of WhyML. It claims to support `finally`. However, from manual inspection of the source it does not seem that `finally` is properly supported. We have concluded this from the fact that in the file `jc_interp.ml`, which is responsible for WhyML code generation, `finally` clauses are ignored. There is also an empty `TODO` comment at this location.

Krakatoa takes a similar approach to the one suggested in this work by encoding Java exceptions into the cleaner exception model of WhyML. However, there are three differences. First, Krakatoa does not do the final translation step into `goto`. Second, if our observations are correct, Krakatoa does not support `finally`. Third, the developers of Krakatoa seem to have missed the insight that the reduction of abrupt termination to exceptions allows for uniform handling of `finally`, resulting in a simpler transformation. Since Krakatoa uses an architecture based on an intermediate representation that is passed through various transformations, we expect they could use this insight effectively.

9.8 VerCors

VerCors has had partial support for inheritance in Java for some time. It has a sparsely documented interface with only a few examples in the test set using the functionality. It has been maintained to keep the few examples in the test set running. However, no structural maintenance has been done. The intended semantics of the functionality was also unclear, but the work by Parkinson & Bierman was mentioned in the documentation.

As far as we know, it used a combination of extended by default predicates from [36], combined with the static/dynamic contract approach of [55]. However, programs that tried to use these features often failed because support was only partially implemented.

Chapter 10

Conclusion

This work aimed to design and document verification support for exceptions and inheritance in Java. To achieve this, we first did a literature study to establish what the state of the art is for verification of exceptions and inheritance. We found that for concurrent languages, the support for exceptions and inheritance is not the rule but the exception. In particular for inheritance the implementation techniques and supported features varied per tool.

We then considered the state of the art approaches and analysed them for incompatibilities with concurrent environments. For exceptions, we discovered an approach that leads to a more modular implementation than what is often implemented in practice. By transforming `break` and `return` to exceptions first, the semantics of `finally` are more modular and the transformation to `goto` is simpler.

For inheritance, we catalogued and made explicit the trade-offs of the known approaches. We concluded it is beneficial for VerCors to combine the APF extension approach with the static/dynamic contract approach. This allows for modular verification of inheritance, allows several useful code patterns, and integrates well with existing VerCors features.

We also conclude that the currently existing techniques for verification of inheritance all have different characteristics and trade-offs. Therefore we believe verification of inheritance in Java is still an open problem.

With this knowledge, we designed a transformation for use in the VerCors tool. The transformations for both inheritance and exceptions make use of the pass-based architecture of VerCors to achieve modular transformations. The proposed design for exceptions was implemented and evaluated using several examples. For inheritance, the proposed design was evaluated by manually applying the proposed transformation to an example.

We conclude that for state-of-the-art verifiers it is currently straightforward to support exceptions and inheritance. Particularly for exceptions, with the current level of existing research and experience, support is only limited by implementation effort. For inheritance support is less obvious, as the state-of-the-art techniques for inheritance still have several shortcomings and trade-offs. However, basic support is straightforward through currently known techniques.

10.1 Future work

We think the following directions of research are interesting for future work:

10.1.1 Formal proof of correctness

Correctness of the transformations presented in this work depend on informal reasoning and textual arguments. A formal proof using an ITP such as Isabelle or Coq would not only be useful for fixing bugs in the suggested approaches, but also increase the confidence in the correctness of the VerCors tool.

10.1.2 Further improving language support

Java language support can be further improved. Lambdas and inner classes are two examples of language features that are also frequently used in commercial software but not yet supported in static verifiers. We believe that providing basic verification support is straightforward. However, providing a convenient verification syntax for these features is not, and requires further research.

Another example of a problematic practical language feature is spurious exceptions, such as `OutOfMemoryError`. These are currently ignored by most static verifiers, including VerCors. However, they can still be problematic in commercial software. Since specifying these exceptions is technically possible, we think research into reducing the notational overhead these kinds of exceptions introduce would be beneficial. This would allow practical verification of programs in the presence of spurious exceptions.

10.1.3 Standard library specification

Java programs frequently use the standard library. It would be useful to have a formal specification of the Java standard library, as it would increase the number of programs that VerCors can verify. We think the two most interesting standard library features are the `InterruptedException` and `Thread.UncaughtExceptionHandler`. These two aspects of the Java standard library will prove challenging, but are important for proving safety and liveness properties.

10.1.4 Improving theory of inheritance

Side-calling in general is heavily relied upon in Java. Further research will have to be done if the capacity for side-calling can be regained while still retaining modelling power and modularity. We expect that APFs will have to be changed to achieve this.

A radical solution would be to side-step the problem and restrict Java inheritance to interfaces. A practical example of this is Classless Java [74]. Research could be done into if this simplified version of Java is more amenable to formal verification.

Finally, the approach presented in this work could also be further optimized for use of interface inheritance.

Bibliography

[1] Wolfgang Ahrendt et al. *Deductive Software Verification - The KeY Book*. New York, NY: Springer Berlin Heidelberg, 2016. ISBN: 978-3-319-49811-9 (cit. on pp. 20, 45, 69, 113).

[2] Wolfgang Ahrendt et al. 'Verification of Smart Contract Business Logic'. In: *Fundamentals of Software Engineering*. Ed. by Hossein Hojjat and Mieke Massink. Springer International Publishing, 2019, pp. 228–243. ISBN: 978-3-030-31517-7 (cit. on p. 45).

[3] Afshin Amighi et al. 'The VerCors project: setting up basecamp'. In: *Proceedings of the sixth workshop on Programming languages meets program verification*. PLPV '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, Jan. 2012. ISBN: 978-1-4503-1125-0. DOI: `10.1145/2103776.2103785` (cit. on p. 17).

[4] Vytautas Astrauskas et al. 'Leveraging rust types for modular specification and verification'. In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), pp. 1–30. ISSN: 24751421. DOI: `10.1145/3360573` (cit. on p. 42).

[5] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte. 'The Spec# Programming System: An Overview'. In: *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Vol. 3362. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 49–69. ISBN: 978-3-540-30569-9. DOI: `10.1007/978-3-540-30569-9_3` (cit. on p. 47).

[6] Mordechai Ben-Ari. *Mathematical logic for computer science*. Springer Science & Business Media, 2012 (cit. on p. 12).

[7] Josh Berdine, Cristiano Calcagno and Peter W. O'Hearn. 'Symbolic Execution with Separation Logic'. In: *Programming Languages and Systems*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 52–68. ISBN: 978-3-540-32247-4. DOI: `10.1007/11575467_5` (cit. on p. 45).

[8] Stefan Blom et al. 'The VerCors Tool Set: Verification of Parallel and Concurrent Software'. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Springer International Publishing, 2017, pp. 102–110. ISBN: 978-3-319-66845-1 (cit. on pp. 37, 45).

[9] François Bobot et al. 'Why3: Shepherd Your Herd of Provers'. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*. Wrocław, Poland, Aug. 2011, pp. 53–64 (cit. on p. 48).

[10] Bernhard F. Brodowsky. 'Translating Scala to SIL'. MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2013. DOI: `10.3929/ethz-a-009908589` (cit. on pp. 44, 95).

[11] Pierre Castéran and Yves Bertot. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004, p. 470 (cit. on p. 50).

[12] A. Cimatti et al. 'NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking'. In: *Proc. International Conference on Computer-Aided Verification (CAV 2002)*. Vol. 2404. LNCS. Copenhagen, Denmark: Springer, July 2002 (cit. on p. 49).

[13] Ernie Cohen et al. 'VCC: A Practical System for Verifying Concurrent C'. In: *Theorem Proving in Higher Order Logics*. Vol. 5674. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42. ISBN: 978-3-642-03359-9. DOI: `10.1007/978-3-642-03359-9_2` (cit. on pp. 47, 48).

[14] David R. Cok. 'OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse'. In: *Electronic Proceedings in Theoretical Computer Science* 149 (Apr. 2014), pp. 79–92. ISSN: 2075-2180. DOI: `10.4204/EPTCS.149.8` (cit. on pp. 20, 45).

[15] Judy Crow et al. 'A Tutorial Introduction to PVS'. In: *Workshop on Industrial-Strength Formal Specification Techniques*. Boca Raton, Florida, Apr. 1995. URL: `http://www.csl.sri.com/papers/wift-tutorial/` (visited on 05/03/2020) (cit. on p. 50).

[16] Carlos E. C. Dantas and Marcelo de Almeida Maia. 'On the Actual Use of Inheritance and Interface in Java Projects: Evolution and Implications'. In: *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*. CASCON '17. USA: IBM Corp., 2017, pp. 151–160 (cit. on p. 2).

[17] Alexandre David et al. 'Uppaal SMC tutorial'. In: *International Journal on Software Tools for Technology Transfer* 17.4 (Aug. 2015), pp. 397–415. ISSN: 1433-2787. DOI: `10.1007/s10009-014-0361-y` (cit. on p. 49).

[18] Thomas Dinsdale-Young et al. 'Caper: Automatic Verification for Fine-grained Concurrency'. In: *Programming Languages and Systems*. Proceedings of the 26th European Symposium on Programming, ESOP 2017. Ed. by Hongseok Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 420–447. ISBN: 978-3-662-54434-1 (cit. on p. 48).

[19] Dino Distefano and Mike Dodds. *How to verify a program with jStar: a tutorial*, p. 23 (cit. on p. 112).

[20] Dino Distefano and Matthew J. Parkinson J. 'jStar: Towards Practical Verification for Java'. In: *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 213–226. ISBN: 978-1-60558-215-3. DOI: `10.1145/1449764.1449782` (cit. on p. 47).

[21] Fabio Drucker. 'Adding Support for Specification Inheritance and Generic Use to the Krakatoa/Why Platform'. In: *Student Honors Theses By Year* (May 2011) (cit. on p. 47).

[22] David J. Eck. *Introduction to Programming Using Java*. Version 8.1. 2019. URL: `http://math.hws.edu/javanotes/` (visited on 24/04/2020) (cit. on p. 7).

[23] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *Computer Aided Verification*. Vol. 10981. Cham: Springer International Publishing, 2018, pp. 596–603. ISBN: 978-3-319-96145-3. DOI: `10.1007/978-3-319-96145-3_33` (cit. on pp. 42, 78, 84, 95, 145).

[24] José Fragoso Santos et al. 'JaVerT: JavaScript Verification Toolchain'. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 50:1–50:33. ISSN: 2475-1421. DOI: `10.1145/3158138` (cit. on p. 45).

[25] Stephen N Freund. 'The costs and benefits of Java bytecode subroutines'. In: *Formal Underpinnings of Java Workshop at OOPSLA 98*. 1998 (cit. on p. 57).

[26] Christian Haack and Clément Hurlin. 'Separation Logic Contracts for a Java-Like Language with Fork/Join'. In: *Algebraic Methodology and Software Technology*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008. ISBN: 978-3-540-79980-1. DOI: `10.1007/978-3-540-79980-1_16` (cit. on pp. 78, 79).

[27] Florian Hahn. 'Rust2Viper: Building a Static Verifier for Rust'. MA thesis. Eidgenössische Technische Hochschule Zürich, Department of Computer Science, 2016, p. 82 (cit. on p. 44).

[28] James Hamilton and Sebastian Danicic. 'An Evaluation of Current Java Bytecode Decompilers'. In: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. 2009, pp. 129–136. DOI: 10.1109/SCAM.2009.24 (cit. on p. 57).

[29] Hans-Dieter A. Hiep et al. *Verifying OpenJDK's LinkedList using KeY*. 2019. arXiv: 1911.04195 [cs.LO] (cit. on p. 45).

[30] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259 (cit. on p. 13).

[31] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual.* Vol. 1003. Addison-Wesley Reading, 2004 (cit. on p. 49).

[32] Black Duck Open Hub. *The Apache Hadoop Open Source Project on Open Hub: Languages Page*. Dec. 2018. URL: https://www.openhub.net/p/Hadoop/analyses/latest/languages_summary (visited on 13/04/2020) (cit. on p. 2).

[33] Black Duck Open Hub. *The Apache Tomcat Open Source Project on Open Hub: Languages Page*. Oct. 2018. URL: https://www.openhub.net/p/tomcat/analyses/latest/languages_summary (visited on 13/04/2020) (cit. on p. 2).

[34] Petr Hudeček. 'Soothsharp: A C#-to-Viper translator'. en. MA thesis. Charles University, Faculty of Mathematics and Physics, 2017, p. 98 (cit. on p. 44).

[35] Marieke Huisman and Bart Jacobs. 'Java Program Verification via a Hoare Logic with Abrupt Termination'. en. In: *Fundamental Approaches to Software Engineering*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 284–303. ISBN: 978-3-540-46428-0 (cit. on p. 15).

[36] Clément Hurlin. 'Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic'. In: (Sept. 2009) (cit. on pp. 5, 78, 79, 81, 114, 137).

[37] Bart Jacobs. 'Provably Live Exception Handling'. In: *Proceedings of the 17th Workshop on Formal Techniques for Java-like Programs*. FTfJP '15. New York, NY, USA: ACM, 2015, 7:1–7:4. ISBN: 978-1-4503-3656-7. DOI: 10.1145/2786536.2786543 (cit. on p. 52).

[38] Bart Jacobs. *Verifast & Java's "finally" clause.* URL: https://groups.google.com/forum/#!topic/verifast/56uhVmdERwA (visited on 05/03/2020) (cit. on p. 109).

[39] Bart Jacobs and Erik Poll. 'Java Program Verification at Nijmegen: Developments and Perspective'. In: *Software Security - Theories and Systems.* Vol. 3233. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 134–153. ISBN: 978-3-540-37621-7. DOI: `10.1007/978-3-540-37621-7_7` (cit. on p. 47).

[40] Bart Jacobs, Jan Smans and Frank Piessens. 'A Quick Tour of the VeriFast Program Verifier'. In: *Programming Languages and Systems.* Vol. 6461. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 304–311. ISBN: 978-3-642-17164-2. DOI: `10.1007/978-3-642-17164-2_21` (cit. on p. 45).

[41] Bill Joy et al. *The Java language specification.* Java SE 7 Edition. Addison-Wesley Reading, 2000 (cit. on pp. 2, 53, 54, 90, 94).

[42] *Key Project - Applications - Program Verification.* `https://www.key-project.org/applications/program-verification/`. [Online; accessed 24-oktober-2019]. 2019 (cit. on p. 45).

[43] Florent Kirchner et al. 'Frama-C: A software analysis perspective'. In: *Formal Aspects of Computing* 27.3 (May 2015), pp. 573–609. ISSN: 0934-5043, 1433-299X. DOI: `10.1007/s00165-014-0326-7` (cit. on p. 44).

[44] Peter Lammich. 'Generating Verified LLVM from Isabelle/HOL'. In: *10th International Conference on Interactive Theorem Proving (ITP 2019).* Ed. by John Harrison, John O'Leary and Andrew Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 22:1–22:19. ISBN: 978-3-95977-122-1. DOI: `10.4230/LIPIcs.ITP.2019.22` (cit. on p. 50).

[45] K. Rustan M. Leino. 'This is Boogie 2'. June 2008. URL: `https://www.microsoft.com/en-us/research/publication/this-is-boogie-2-2/` (visited on 25/03/2020) (cit. on p. 48).

[46] Barbara H. Liskov and Jeannette M. Wing. 'A behavioral notion of subtyping'. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: `10.1145/197320.197383` (cit. on p. 11).

[47] Claude Marché, Christine Paulin-Mohring and Xavier Urbain. 'The Krakatoa tool for certification of Java/JavaCard programs annotated in JML'. In: *Journal of Logic and Algebraic Programming* 58.1-2 (2004), pp. 89–106 (cit. on pp. 9, 47).

[48] Tobias Nipkow. *Programming and proving in Isabelle/HOL.* 2013 (cit. on p. 50).

[49] Peter O'Hearn. 'Separation logic'. In: *Communications of the ACM* 62.2 (Jan. 2019), pp. 86–95. ISSN: 00010782. DOI: `10.1145/3211968` (cit. on p. 22).

[50] Peter O'Hearn, John Reynolds and Hongseok Yang. 'Local Reasoning about Programs that Alter Data Structures'. In: *Computer Science Logic*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 1–19. ISBN: 978-3-540-44802-0. DOI: `10.1007/3-540-44802-0_1` (cit. on p. 22).

[51] Peter W. O'Hearn. 'Resources, concurrency, and local reasoning'. In: *Theoretical Computer Science*. Festschrift for John C. Reynolds's 70th birthday 375.1 (May 2007), pp. 271–307. ISSN: 0304-3975. DOI: `10.1016/j.tcs.2006.12.035`. (Visited on 06/09/2019) (cit. on p. 22).

[52] Haidar Osman et al. 'On the evolution of exception usage in Java projects'. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Feb. 2017, pp. 422–426. DOI: `10.1109/SANER.2017.7884646` (cit. on p. 2).

[53] Matthew Parkinson and Gavin Bierman. 'Separation Logic for Object-Oriented Programming'. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 366–406. ISBN: 978-3-642-36946-9. DOI: `10.1007/978-3-642-36946-9_13` (cit. on pp. 5, 81, 86, 93, 111).

[54] Matthew J. Parkinson. *Local reasoning for Java*. Tech. rep. UCAM-CL-TR-654. University of Cambridge, Computer Laboratory, 2005 (cit. on pp. 5, 10, 25, 78, 111, 137).

[55] Matthew J. Parkinson and Gavin M. Bierman. 'Separation Logic, Abstraction and Inheritance'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. New York, NY, USA: ACM, 2008, pp. 75–86. ISBN: 978-1-59593-689-9. DOI: `10.1145/1328438.1328451` (cit. on pp. 81, 83, 114).

[56] Github Verifast repository. *ArrayList.java*. URL: `https://github.com/verifast/verifast/blob/f5e7497c2ef5fe9ada188515bfeb323f395e765b/examples/java/ArrayList.java` (visited on 02/04/2020) (cit. on p. 111).

[57] Guido van Rossum, Jukka Lehtosalo and Łukasz Langa. *Type Hints*. PEP 484. 2014. URL: `https://www.python.org/dev/peps/pep-0484/` (visited on 24/03/2020) (cit. on pp. 42, 109).

[58] Bob Rubbens. *Exceptions and abrupt termination example*. URL: `https://github.com/bobismijnnaam/vercors/blob/bd21b92c2ee450ae85caaa542cbd8db3c79c47ba/examples/abrupt/ExceptionsAndAbrupt.java` (visited on 08/05/2020) (cit. on p. 71).

[59] Bob Rubbens. *Exceptions branch of forked VerCors repository*. URL: `https://github.com/bobismijnnaam/vercors/tree/bd21b92c2ee450ae85caaa542cbd8db3c79c47ba` (visited on 16/05/2020) (cit. on p. 68).

[60] Bob Rubbens. *Incomplete encoding of break.* Nagini GitHub repository. URL: `https://github.com/marcoeilers/nagini/issues/141` (visited on 03/04/2020) (cit. on p. 108).

[61] Bob Rubbens. *KeY Abrupt Termination Challenge example code, adapted for VerCors.* URL: `https://github.com/bobismijnnaam/vercors/blob/bd21b92c2ee450ae85caaa542cbd8db3c79c47ba/examples/abrupt/KeYAbruptTerminationChallenge.java` (visited on 08/05/2020) (cit. on p. 69).

[62] Ö.F.O. Şakar. 'Extending support for axiomatic data types in VerCors'. MA thesis. Apr. 2020. URL: `http://essay.utwente.nl/80892/` (visited on 15/05/2020) (cit. on p. 95).

[63] César Santos, Francisco Martins and Vasco Thudichum Vasconcelos. 'Deductive Verification of Parallel Programs Using Why3'. In: *Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015.* 2015, pp. 128–142. DOI: `10.4204/EPTCS.189.11` (cit. on p. 48).

[64] Demóstenes Sena et al. 'Understanding the exception handling strategies of Java libraries: an empirical study'. In: *Proceedings of the 13th International Conference on Mining Software Repositories.* MSR '16. Austin, Texas: Association for Computing Machinery, May 2016, pp. 212–222. ISBN: 978-1-4503-4186-8. DOI: `10.1145/2901739.2901757` (cit. on p. 2).

[65] Jan Smans, Bart Jacobs and Frank Piessens. 'VeriFast for Java: A Tutorial'. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification.* Vol. 7850. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 407–442. ISBN: 978-3-642-36946-9. DOI: `10.1007/978-3-642-36946-9_14` (cit. on pp. 45, 78, 111).

[66] Andrei Ştefănescu et al. 'All-Path Reachability Logic'. In: *Rewriting and Typed Lambda Calculi.* Ed. by Gilles Dowek. Springer International Publishing, 2014, pp. 425–440. ISBN: 978-3-319-08918-8 (cit. on p. 46).

[67] Andrei Ştefănescu et al. 'Semantics-based Program Verifiers for All Languages'. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications.* OOPSLA 2016. New York, NY, USA: ACM, 2016, pp. 74–91. ISBN: 978-1-4503-4444-9. DOI: `10.1145/2983990.2984027`. URL: `http://doi.acm.org/10.1145/2983990.2984027` (cit. on p. 46).

[68] Dominic Steinhöfel and Nathan Wasser. 'A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows'. In: *Integrated Formal Methods.* Vol. 10510. Cham: Springer International Publishing, 2017, pp. 279–294. DOI: `10.1007/978-3-319-66845-1_18` (cit. on p. 113).

[69] The Mypy Team. *Mypy: Optional Static Typing for Python.* URL: `http://mypy-lang.org/` (visited on 03/05/2020) (cit. on p. 109).

[70] The VerCors Team. *VerCors Homepage.* URL: `https://vercors.ewi.utwente.nl/` (visited on 09/05/2020) (cit. on p. 2).

[71] TIOBE Software BV. *TIOBE Index.* URL: `https://www.tiobe.com/tiobe-index/` (visited on 27/03/2020) (cit. on p. 1).

[72] University of Twente, FMT group. *Home - utwente_fmt/vercors Wiki.* URL: `https://github.com/utwente-fmt/vercors/wiki` (visited on 21/04/2020) (cit. on p. 14).

[73] M Mitchell Waldrop. 'The chips are down for Moore's law'. In: *Nature News* 530.7589 (2016), p. 144 (cit. on p. 1).

[74] Yanlin Wang et al. 'Classless Java'. In: *ACM SIGPLAN Notices* 52.3 (Oct. 2016), pp. 14–24. ISSN: 0362-1340. DOI: `10.1145/3093335.2993238` (cit. on p. 117).

Appendices

Appendix A

List of Rewrite Rules

Appendix B

Code duplication increases verification time

In this appendix we will discuss a small informal experiment that shows that code duplication indeed causes increased verification time. First we will discuss the problem, followed by the design of the experiment. Then we will discuss the results, and will conclude with a brief explanation of the source code format used in the experiment.

B.1 Problem statement

Some approaches in this work, such as the inlining approach suggested in Section 7.1.5, duplicate statements. By applying this transformation, the final result of the program remains the same, but the number of statements in the program submitted to Viper will increase. Currently, VerCors does not have facilities to mark these statements as "duplicate" in any way to prevent duplicate proof obligations. Therefore, our hypothesis is: duplication of program elements will increase verification time.

B.2 Method

VerCors does modular verification. This means that effectively methods are considered in isolation. Therefore we want to measure the effect of duplication at both levels: at the level of one isolated unit, and the level of multiple isolated units. To achieve this, we will duplicate a particular statement, and also whole methods. This will give insight if VerCors can reuse proofs from related statements, and also if VerCors can reuse proofs from similar methods.

The program that we will test will need to have a noticeable but short verification time. It is included in Section B.4. For this we picked an incomplete

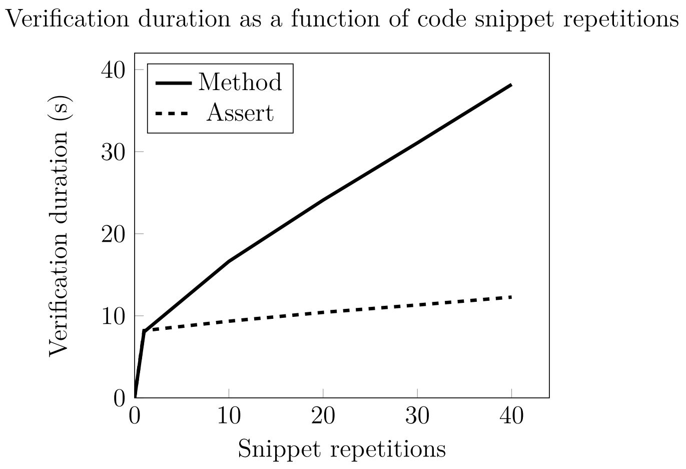


Figure B.1: The continuous line shows the verification time when only the method is duplicated. The dashed line shows the verification time when only the assert is duplicated. Shown up to 40 repetitions.

annotated implementation of the merge sort algorithm from our personal archives. Verification of this program takes 8.6 seconds on average. The specific assert that was picked from this program contributes 0.4 seconds to the total verification time on average.

For each duplication, verification time will be measured. Verification time will be measured by doing one warm-up run, and then 5 runs of which verification time will be averaged. The assert and method will be duplicated independently. This approach should show the impact of duplicating statements and methods. If VerCors can reuse parts of the proof, the duration should not increase linearly or even stay the same. If VerCors cannot reuse parts of the proofs, the growth of verification duration should be a straight line.

The experiment was run on a Thinkpad P50 with an Intel i7-6700HQ CPU at 2.60GHz. The commit for VerCors used is:

```
f6f45dc1717a99fb3b6df6fefb1de831d5e72b3d
```

B.3 Results & Discussion

The results for the first 40 runs are shown in Figure B.1. The results for all 400 runs are shown in Figure B.2.

It is shown in Figure B.1 that verification incurs a substantial start cost of 8 seconds. However, as asserts and methods are duplicated, the verification time

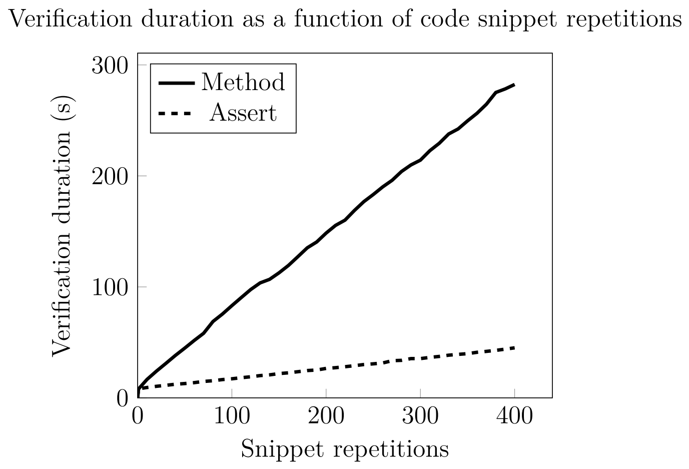


Figure B.2: The continuous line shows the verification time when only the method is duplicated. The dashed line shows the verification time when only the assert is duplicated. Shown up to 400 repetitions.

grows in a linear progression less steep than an extrapolation of the initial verification time.

The data suggest that VerCors can reuse proofs from earlier assertions, as adding assertions adds less time than having only one of the assertion at the beginning. Furthermore, it also suggests VerCors can reuse proofs from earlier method definitions, as that line also grows less steep than the initial growth. However, the proofs from earlier statements is not zero: duplicating statements and methods clearly increase verification time. Therefore, we conclude that while the impact of repeating code is not big, it is not the case that repeated code does not cause extra overhead: duplication of statements and methods will increase verification time.

B.4 Source code used in the experiment

Below the source of the program used for testing is presented. The sections that are repeated are delineated by comments of the form

```
// performance duplication section start
.. regular code ...
// performance duplication section end.
```

These comments are picked up by a script, which repeats the sections if needed and then runs VerCors on the duplicated program.

The method that is repeated begins on line 18. The assert that is duplicated is on line 58.

```
 1  class MyClass {
 2    ensures \result <= a && \result <= b;
 3    pure int min(int a, int b) = a < b ? a : b;
 4
 5    requires (\forall int i; 0 <= i && i < |xs|;
 6        (\forall int j; 0 <= j && j < i; xs[j] <= xs[i]));
 7    requires (\forall int i; 0 <= i && i < |ys|;
 8        (\forall int j; 0 <= j && j < i; ys[j] <= ys[i]));
 9    ensures |\result| == (|xs| + |ys|);
10    ensures (|xs| > 0 && |ys| == 0) ==> \result == xs;
11    ensures (|xs| == 0 && |ys| > 0) ==> \result == ys;
12    ensures (|xs| == 0 && |ys| == 0) ==> |\result| == 0;
13    ensures (|xs| > 0 && |ys| > 0) ==> \result[0] == min(xs[0], ys[0]);
14    ensures (\forall int i; 0 <= i && i < |\result|;
15        (\forall int j; 0 <= j && j < i; \result[j] <= \result[i]));
16    seq<int> mergeSeq(seq<int> xs, seq<int> ys);
17
18    // ----- performance duplication method start
19    ensures (\forall int i; 0 <= i && i < |\result|;
20        (\forall int j; 0 <= j && j < i; \result[j] <= \result[i]));
21    seq<int> mergeSortXXX(seq<int> xs) { // X X X is replaced with number!
22      if (|xs| <= 1) {
23        assert (\forall int i; 0 <= i && i < |xs|;
24          (\forall int j; 0 <= j && j < i; xs[j] < xs[i]));
25        return xs;
26      } else {
27        int len = |xs|;
28        int pivot = len / 2;
29
30        seq<int> ys = seq<int>{};
31        loop_invariant 0 <= i && i <= pivot;
32        loop_invariant |ys| == i;
33        loop_invariant (\forall int k; 0 <= k && k < i; xs[k] == ys[k]);
34        for (int i = 0; i < pivot; i = i + 1) {
35          ys = ys + seq<int>{xs[i]};
36        }
37
38        assert (\forall int k; 0 <= k && k < pivot; xs[k] == ys[k]);
39
40        seq<int> zs = seq<int>{};
41        loop_invariant pivot <= j && j <= |xs|;
42        loop_invariant |zs| == (j - pivot);
43        loop_invariant (\forall int k; pivot <= k && k < j; xs[k] == zs[k - pivot
              ]);
44        for (int j = pivot; j < |xs|; j = j + 1) {
45          zs = zs + seq<int>{xs[j]};
```

```
      }

      assert (\forall int k; pivot <= k && k < |xs|; xs[k] == zs[k - pivot]);

      assert xs == (ys + zs);

      seq<int> as = mergeSortXXX(ys);
      seq<int> bs = mergeSortXXX(zs);

      assert (\forall int i; 0 <= i && i < |as|;
        (\forall int j; 0 <= j && j < i; as[j] <= as[i]));

      // ----- performance duplication assert start
      assert (\forall int i; 0 <= i && i < |bs|;
        (\forall int j; 0 <= j && j < i; bs[j] <= bs[i]));
      // ----- performance duplication assert end

      return mergeSeq(as, bs);
    }
  }
  // ----- performance duplication method end
}
```

Appendix C

Wrapped APF issue

In Nagini, when an APF is wrapped in a predicate and used in the postcondition of a constructor, the `Fold` is not included when inlining the constructor in a child constructor. Hence, any `Fold`s depending on earlier `Fold`s of APFs unexpectedly fail.

Clearly the inlining of the method fails here because a fold is missing. Hence, Child can never implement its constructor. However, the right way to go would be to refactor, since this program structure is not useful. This would solve the issue.

A copy of the error message output by Nagini when this file is verified is included in the listing at line 46.

Listing C.1: Demonstrates verification failure when an APF is wrapped in a predicate and used in the postcondition of a constructor.

```
from nagini_contracts.contracts import *

# First activate the nagini environment:
# $ source ~/UNSAFE/dummy/nagini_test/nagini_env/bin/activate
# Then run:
# $ nagini apf_inlining_practicality_issue.py

class Parent:
    @Predicate
    def p(self, x: int) -> bool:
        return Acc(self.v) and self.v == x

    def __init__(self) -> None:
        Ensures(wrapper_p(self, 0))

        self.v = 0
        Fold(self.p(0))
        Fold(wrapper_p(self, 0))

@Predicate
def wrapper_p(self: Parent, x: int) -> bool:
    return self.p(x)

class Child(Parent):
    def __init__(self) -> None:
        Ensures(child_p(self, 0))

        # Error at the following line because
        # the second fold in super.__init__ fails
        super(Child, self).__init__()

        self.bak = 0

        # Hence the following lines can never succeed
        Fold(wrapper_p(self, 0))
        Fold(child_p(self, 0))

@Predicate
def child_p(self: Child, x: int) -> bool:
    return Acc(self.bak) and self.bak == x and wrapper_p(self, x)

def main() -> None:
    parent = Parent()
    Assert(wrapper_p(parent, 0))

# Error:
```

```
#
# (nagini_env) bobe@lapbobe:~/UNSAFE/Studie/thesis/python_programs
# $ nagini apf_inlining_practicality_issue.py
# Verification failed
# Errors:
# Fold might fail. There might be insufficient permission to access
    self.p(x). (apf_inlining_practicality_issue.py@18.8, via static
    call at apf_inlining_practicality_issue.py@30.8)
# Verification took 8.16 seconds.
```

Appendix D

Informal Inheritance Semantics

D.1 Introduction

This appendix presents an informal semantics for the inheritance approach and new syntax suggested in Chapter 8. It is not intended to be formal or leading over the information presented in Chapter 8, but meant to accompany Chapter 8 and supply more information in situations where Chapter 8 fails to convey the proper idea or is incomplete.

Ideally it would be used to document and further develop inheritance support in VerCors.

D.2 Notation

Most of the notation used in this appendix should be similar to notation used in logic in general, as well as the works of [54] and [36]. However, for completeness, we include an overview of the notation used.

$o : C$ means *the static type of* o *is equal to class* C.
$C <: D$ means C `extends/implements` D.
Elements in code font [`between brackets`] are optional.
$P[\bar{e}/\bar{x}]$ means P *where every* x_i *is replaced with* e_i.
$\bar{x}$ means for a sequence of n variables $x_0, \cdots, x_{n-1}$.
$|\bar{x}|$ means *length of sequence* $\bar{x}$.
$\bar{x} \subseteq \bar{y}$ means $|\bar{x}| \leq |\bar{y}| \wedge \forall i \in [0, |\bar{x}|) \,.\, x_i = y_i$
$\mathit{static}(F)$ replaces every occurrence of $\mathit{this.p}(\bar{e})$ with $\mathit{this.p}@C(\bar{e})$ where $\mathit{this} : C$.
$\mathit{dynamic}(F)$ does not change the contract. It is included for symmetry with $\mathit{static}(F)$.

`public`/`private` qualifiers are ignored on predicates. Predicates can be folded/unfolded by any class. APF entries can only be folded/unfolded by the class defining the specific APF entry. This is chosen this way because it is the current state of VerCors. In the future, it is probably a good idea to respect the accessibility qualifiers of predicates in a useful way.

To keep the rules for method and constructor calls brief, it holds that for any rule correlating a call and a definition the arity of the definition is the same as the arity of the call. This does not include `given` and `yields`: in the proof steps proving compatibility between a sub-method and a super-method, `given` parameters of the sub-method can be given a value through `given-widen` and unfolding of predicates. Parameters can also be left unused if the overridden APF of the subclass has less parameters.

Ghost-code or specification delimiters such as `//@` or `/*@ ... @*/` are not included in the rules, but should be used in Java code where appropriate.

D.3 Rules

Method calls

$$\frac{o : C \quad C <: D \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public final}\ U\ D.m(\bar{X}\ \bar{x})}{\{static(F)\}\ o.m(\bar{e})\ \{static(G)\}}\ \text{(FinalCall)}$$

$$\frac{o : C \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ U\ C.m(\bar{X}\ \bar{x})}{\{dynamic(F)\}\ o.m(\bar{e})\ \{dynamic(G)\}}\ \text{(DynamicCall)}$$

$$\frac{\texttt{this} : C \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{private [final]}\ U\ C.m(\bar{X}\ \bar{x})}{\{static(F)\}\ \texttt{this}.m(\bar{e})\ \{static(G)\}}\ \text{(PrivateCall)}$$

Constructor calls

$$\frac{\texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public [final]}\ U\ C(\bar{X}\ \bar{x})}{\{static(F)\}\ \texttt{new}\ C(\bar{e})\ \{static(G) * \texttt{\textbackslash result}.getClass() == C.\texttt{class}\}}\ \text{(New)}$$

`super` calls

$$\frac{\textit{this} : C \quad C\ \texttt{extends}\ D \qquad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ \ \texttt{[final]}\ U\ D(\bar{X}\ \bar{x})}{\{\textit{static}(F)\}\ \texttt{super}(\bar{e})\ \{\textit{static}(G)\}}\ \text{(SuperConstructor)}$$

$$\frac{C\ \texttt{extends}\ D \qquad \textit{this} : C \quad \texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ \ \texttt{[final]}\ U\ D.m(\bar{X}\ \bar{x})}{\{\textit{static}(F)\}\ \texttt{super}.m(\bar{e})\ \{\textit{static}(G)\}}\ \text{(SuperCall)}$$

Method declaration

Declare a new method:

$$\frac{C\ \textit{extends}\ D \quad m \notin D \quad \{\textit{static}(F)\}\ \bar{c}\ \{\textit{static}(G)\}}{\texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ \ \texttt{[final]}\ U\ C.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \}}\ \text{(NewMethod)}$$

Override an existing method:

$$\frac{\begin{array}{ll} & \texttt{requires}\ F';\ \texttt{ensures}\ G';\ \texttt{public}\ U\ \ D.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \} \\ \{\textit{static}(F)\}\ \bar{c}\ \{\textit{static}(G)\} & \qquad \textit{dynamic}(F') \mathrel{-\!\!*} \textit{dynamic}(F) \\ \quad C\ \textit{extends}\ D & \qquad \textit{dynamic}(G) \mathrel{-\!\!*} \textit{dynamic}(G') \end{array}}{\texttt{requires}\ F;\ \texttt{ensures}\ G;\ \texttt{public}\ \ \texttt{[final]}\ U\ C.m(\bar{X}\ \bar{x})\ \{\ \bar{c}\ \}}\ \text{(OverrideMethod)}$$

Using abstract predicates

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \notin D \quad \texttt{final}\ \ \texttt{resource}\ C.p(\bar{X}\bar{x}) = P}{\{o.p(\bar{e})\}\ \texttt{unfold}\ o.p(\bar{e})\ \{P[\bar{e}/\bar{x}]\}}\ \text{(UnfoldPredicate)}$$

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \notin D \quad \texttt{final resource}\ C.p(\bar{X}\bar{x}) = P}{\{P[\bar{e}/\bar{x}]\}\ \texttt{fold}\ o.p(\bar{e})\ \{o.p(\bar{e})\}}\ \text{(FoldPredicate)}$$

Defining abstract predicates

$$\frac{C\ \texttt{extends}\ D \quad p \notin D}{\texttt{final resource}\ C.p(\bar{X}\bar{x}) = P}\ \text{(NewPredicate)}$$

Using Abstract Predicate Family Entries

$$\frac{\begin{array}{c} o : C \quad C\ \texttt{extends}\ D \quad \texttt{resource}\ D.p(\bar{Y}\bar{y}) = Q \quad \bar{f} \subseteq \bar{e} \\ \texttt{[final] resource}\ C.p(\bar{X}\bar{x}) = P \end{array}}{\{o.p@C(\bar{e})\}\ \texttt{unfold}\ o.p@C(\bar{e})\ \{P[\bar{e}/\bar{x}] * o.p@D(\bar{f})\}}\ \text{(UnfoldEntry)}$$

$$\frac{\begin{array}{c} o : C \quad C\ \texttt{extends}\ D \quad \texttt{resource}\ D.p(\bar{Y}\bar{y}) = Q \quad \bar{f} \subseteq \bar{e} \\ \texttt{[final] resource}\ C.p(\bar{X}\bar{x}) = P \end{array}}{\{P[\bar{e}/\bar{x}] * o.p@D(\bar{f})\}\ \texttt{fold}\ o.p@C(\bar{e})\ \{o.p@C(\bar{e})\}}\ \text{(FoldEntry)}$$

Defining Abstract Predicate Families

$$\frac{C\ \texttt{extends}\ D \quad \texttt{resource}\ D.p(\bar{Y}\bar{y}) = Q \quad |\bar{x}| \geq |\bar{y}|}{\texttt{[final] resource}\ C.p(\bar{X}\bar{x}) = P}\ \text{(ExtendFamily)}$$

$$\frac{C\ \texttt{extends}\ D \quad p \notin D}{\texttt{resource}\ C.p(\bar{X}\bar{x}) = P}\ \text{(NewFamily)}$$

$$\frac{C\ \texttt{extends}\ D \quad \texttt{resource}\ D.p(\bar{Y}\bar{y}) = P \quad p \notin C \quad |\bar{x}| = |\bar{y}|}{\texttt{resource}\ C.p(\bar{X}\bar{x}) = \texttt{true}}\ \text{(ImplicitFamily)}$$

Using Abstract Predicate Families

Read-only access to predicate entries:

$$\frac{\texttt{resource}\ C.p(\bar{X}\bar{x}) = P}{\begin{array}{c}\{o\ \texttt{instanceof}\ C * o.p(\bar{e})\}\\ \texttt{extract}\ o.p@C(\bar{e})\\ \{o\ \texttt{instanceof}\ C * o.p@C(\bar{e}) * o.p@C(\bar{e}) \mathrel{-\!\!*} o.p(\bar{e})\}\end{array}}\ \text{(Extract)}$$

Access when dynamic type is known:

$$\frac{\texttt{resource}\ C.p(\bar{X}\bar{x}) = P}{\begin{array}{c}\{o.getClass() == C.class * o.p(\bar{e})\}\\ \texttt{unfold}\ o.p(\bar{e})\ \texttt{at}\ C\\ \{o.getClass() == C.class * o.p@C(\bar{e})\}\end{array}}\ \text{(UnfoldFamily)}$$

$$\frac{\texttt{resource}\ C.p(\bar{X}\bar{x}) = P}{\begin{array}{c}\{o.getClass() == C.class * o.p@C(\bar{e})\}\\ \texttt{fold}\ o.p(\bar{e})\ \texttt{at}\ C\\ \{o.getClass() == C.class * o.p(\bar{e})\}\end{array}}\ \text{(FoldFamily)}$$

Access when dynamic type is statically known:

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \in D \quad \texttt{final resource}\ C.p(\bar{X}\bar{x}) = P}{\{o.p(\bar{e})\}\ \texttt{unfold}\ o.p(\bar{e})\ \{o.p@C(\bar{e})\}}\ \text{(UnfoldFinalFamily)}$$

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \in D \quad \texttt{final resource}\ C.p(\bar{X}\bar{x}) = P}{\{o.p@C(\bar{e})\}\ \texttt{fold}\ o.p(\bar{e})\ \{o.p(\bar{e})\}}\ \text{(FoldFinalFamily)}$$

Changing arity of abstract predicates:

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \in D \quad \texttt{[final] resource}\ C.p(\bar{X}\bar{x}) = P}{\{o.p(\bar{e})\}\ \texttt{given}\ V\ v\ \texttt{widen}\ o.p(\bar{e})\ \{\exists v : V.\ o.p(\bar{e}, v)\}}\ \text{(Widen)}$$

$$\frac{o : C \quad C\ \texttt{extends}\ D \quad p \in D \quad \texttt{[final] resource}\ C.p(\bar{X}\bar{x}) = P \quad \bar{e} = \bar{f}, v}{\{o.p(\bar{e})\}\ \texttt{narrow}\ o.p(\bar{e})\ \{o.p(\bar{f})\}}\ \text{(Narrow)}$$

Appendix E

Method override mandatory

In the code sample on the next page, allowing inheritance of the method `inc` would be unsound, as the definition of the APF `p` in `Child` requires `my_v` to be incremented whenever `v` is incremented. Therefore, `inc` can only ever be inherited if the inherited method also somehow increments `my_v`. For this, an implementation is needed, and hence automatic inheritance is impossible. In the code, the static/dynamic approach is assumed. Note that the `inc` method in `Child` is not an actual method, but an example of what the proof of the inherited method would look like.

Listing E.1: Example of a program where automatic inheritance would be harmful.

```
class Parent {
  public int v;

  resource p(int x) = Perm(v, 1) ** v == x;

  given int x;
  requires p(x);
  ensures p(x+1);
  public void inc() {
    unfold p@Parent(x);
    v = v + 1;
    fold p@Parent(x + 1);
  }
}

class Child extends Parent {
  public int my_v;

  resource p(int x) = Perm(my_v, 1) ** my_v == x;

  // Inherited method
  given int x; // Inherited
  requires p(x); // Inherited
  ensures p(x+1); // Inherited
  public void inc() {
    assert p@Child(x)
    unfold p@Child(x)
    assert p@Parent(x) ** my_v == x;
    super.inc();
    assert p@Parent(x+1) ** my_v == x;
    assert my_v == x + 1; // Fails
    fold p@Child(x+1); // Cannot succeed, as previous assert fails
  }
}
```

Appendix F

Java Type ADT

In this appendix we show an example of a type ADT that can model the type system of Java in Silver. This type ADT was first used by the Nagini verifier [23], which is designed and implemented by Marco Eilers. We believe that this ADT is flexible enough to be used for modelling the Java type system as well.

The type ADT mainly uses the following two functions: `extends` and `issubtype`. The `extends` function is used whenever inheritance appears syntactically: for example, when a class indicates it `extends` another class. The `issubtype` is supposed to be derived from all applications of `extends` by transitivity and reflection. For example, if `A extends B`, and `B extends C`, then it is not the case that `extends(A, C)`, but it is the case that `issubtype(A, C)`.

The ADT uses several axioms to model the type system. The first group of axioms ensures the subtyping relation is established and propagated. This group starts on line 19 and ends on line 32. It has axioms that state facts such as transitivity, reflexivity, and that any type is a subtype of Object.

The second group of axioms deals with deriving facts about when two types are not subtypes. This group starts on line 44 and ends on line 59. It has axioms that state facts such as:

- Two types that extend the same supertype cannot be subtypes.
- If a type is different from an other type and a subtype of that type, the other type is not a subtype of that type.

At the end of the listing on line 70, several statements are included that show how the type ADT is used in Silver code.

```
domain TYPE {

  function extends(sub: TYPE, super: TYPE): Bool

  function issubtype(sub: TYPE, super: TYPE): Bool

  function isnotsubtype(sub: TYPE, super: TYPE): Bool

  function typeof(obj: Ref): TYPE

  unique function Object(): TYPE

  unique function NullType(): TYPE

  unique function Node(): TYPE

  unique function Leaf(): TYPE

  // Subtyping is transitive
  axiom issubtype_transitivity {
    (forall sub: TYPE, middle: TYPE, super: TYPE :: { issubtype(sub,
        middle),issubtype(middle, super) } issubtype(sub, middle) &&
        issubtype(middle, super) ==> issubtype(sub, super))
  }

  // Any type is a subtype of itself
  axiom issubtype_reflexivity {
    (forall type_: TYPE :: { issubtype(type_, type_) } issubtype(
        type_, type_))
  }

  // Extends establishes subtype relation
  axiom extends_implies_subtype {
    (forall sub: TYPE, sub2: TYPE :: { extends(sub, sub2) } extends(
        sub, sub2) ==> issubtype(sub, sub2))
  }

  // Only null is subtype of the Null type
  axiom null_nonetype {
    (forall r: Ref :: { typeof(r) } issubtype(typeof(r), NullType())
        == (r == null))
  }

  // Any type subtypes object
  axiom issubtype_Object {
    (forall type_: TYPE :: { issubtype(type_, Object()) } issubtype(
        type_, Object()))
  }

  // If a and b extend c, a is not a subtype of b and vice versa
  axiom issubtype_exclusion {
    (forall sub: TYPE, sub2: TYPE, super: TYPE :: { extends(sub,
        super),extends(sub2, super) } extends(sub, super) && extends(
        sub2, super) && sub != sub2 ==> isnotsubtype(sub, sub2) &&
        isnotsubtype(sub2, sub))
  }

  // If a is a subtype of b, b is not a subtype of a
  axiom issubtype_exclusion_2 {
    (forall sub: TYPE, super: TYPE :: { issubtype(sub, super) } {
        issubtype(super, sub) }
    issubtype(sub, super) && sub != super ==> !issubtype(super, sub))
  }

  // if a subtypes b, but b does not subtype c, a is not a subtype of
      c
  axiom issubtype_exclusion_propagation {
    (forall sub: TYPE, middle: TYPE, super: TYPE :: { issubtype(sub,
        middle),isnotsubtype(middle, super) }
    issubtype(sub, middle) && isnotsubtype(middle, super) ==> !
        issubtype(sub, super))
  }

  axiom subtype_Node {
    extends(Node(), Object())
  }

  axiom subtype_Leaf {
    extends(Leaf(), Object())
  }
}

method foo() {
  var x: Ref;
  x := new();
  inhale typeof(x) == Node()

  assert issubtype(typeof(x), Node())
  assert typeof(x) != Leaf()
  assert issubtype(Node(), Object())
  assert !issubtype(Node(), Leaf())
  assert !(issubtype(typeof(x), Leaf()))
}
```

Appendix G

Cell/ReCell Full Program

On the next page, we list the full of the example evaluated in Section 8.4.

```
class Cell {
  int val;
  //@ resource state(int x) = Perm(val, 1\1) ** x == val;

  //@ requires state(x);
  //@ ensures state(x) ** \result == x;
  int get() {
    //@ unfold state@Cell(x);
    return val;
    //@ fold state@Cell(x);
  }

  //@ requires state(oldval);
  //@ ensures state(newVal);
  void set(int newVal) {
    //@ unfold state@Cell(oldVal);
    val = newVal;
    //@ fold state@Cell(newVal);
  }
}

class ReCell extends Cell {
  int bak;
  //@ resource state(int x, int y) = Perm(bak, 1\1) ** y == bak;

  //@ requires state(x, y);
  //@ ensures state(x, y) ** \result == x;
  /*@ with {
      given (int y) widen state(x);
    }; @*/
  /*@ then {
      narrow state(x, y);
    }; @*/
  int get() {
    //@ unfold state@ReCell(x, y);
    return super.get();
    //@ fold state@ReCell(x, y);
  }

  //@ requires state(oldVal, oldBak);
  //@ ensures state(newVal, oldVal);
  /*@ with {
        given (int oldBak) widen state(oldVal);
      }; @*/
  /*@ then {
        narrow state(newVal, oldval);
      }; @*/
  void set(int newVal) {
    //@ unfold state@ReCell(x, y);
    bak = super.get();
    super.set(newVal);
    //@ fold state@ReCell(x, y);
  }
}
```