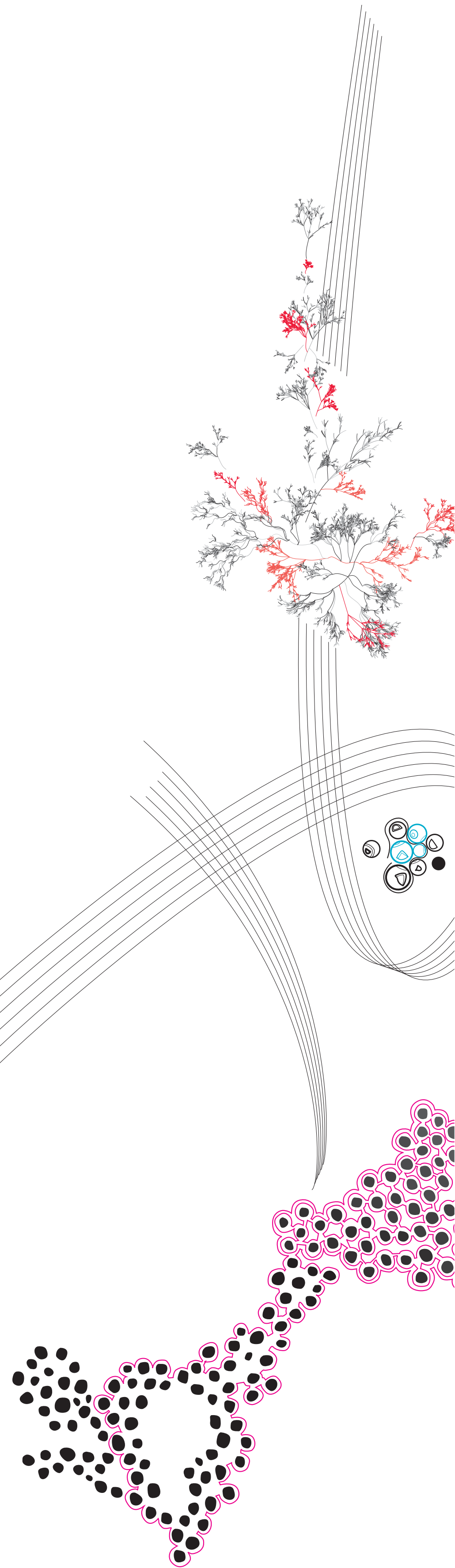
Master Thesis Computer Science



VCLLVM: A Transformation Tool for LLVM IR programs to aid Deductive Verification

D.H.M.A. van Oorschot BSc

Graduation Committee:
prof.dr. M. Huisman
dr.ing. K.H. Chen
Ö.F.O. Şakar MSc

July, 2023

Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

VCLLVM: A Transformation Tool for LLVM IR programs to aid Deductive Verification

D.H.M.A. van Oorschot

July, 2023

Abstract

Errors in computer programs are as old as the discipline of computer programming itself. While the nature of these errors has changed over time, the fact remains that computer bugs will never be eliminated. For this reason, research into software verification techniques is essential for the future of software development. Verification techniques come in many forms.

VerCors is a verification toolset that uses deductive verification for languages such as Java, C, and its own internal PVL (Prototypal Verification Language). However, developing and maintaining support for a language adds significant development overhead to the project, as semantic models must be defined and updated separately for each language.

This thesis presents VCLLVM, a project that adds LLVM IR as a supported language to the VerCors toolset. Having LLVM IR supported as a language opens future possibilities for verifying any language that is compiled through the LLVM infrastructure. Moreover, the tandem of VCLLVM/VerCors would bring novel research to the field of software verification as to the best of our knowledge, it is the first deductive verifier for LLVM IR.

Keywords: Static Verification, Deductive Verification, LLVM, Permission-Based Separation Logic

Contents

1 Introduction **1**

2 Background **5**
2.1 VerCors 5
2.1.1 VerCors Architecture 5
2.1.2 Permission-Based Separation Logic 6
2.1.2.1 Hoare Logic 6
2.1.2.2 Permission-Based Separation Logic 9
2.1.3 VerCors specification language 14
2.1.3.1 Method Contracts 14
2.1.3.2 Permissions 15
2.1.3.3 Arrays and Loops 17
2.2 The LLVM Project 19
2.2.1 LLVM IR 20
2.2.1.1 Representation Formats 20
2.2.1.2 Component Hierarchy 21
2.2.1.3 Features of LLVM IR 21

3 Problem Statement **31**
3.1 Research Question 31
3.2 Challenges 31
3.2.1 Challenge 1: Instability of LLVM IR 31
3.2.2 Challenge 2: Parsing Verification Specifications 32
3.2.3 Challenge 3: Origin of User Errors 32
3.2.4 Challenge 4: VerCors and Compiler Intermediate Representations . . . 32
3.2.5 Challenge 5: Control Flow Structuring 33
3.2.6 Challenge 6: LLVM Concurrency Model 33
3.2.7 Challenge 7: `undef` and `poison` Types 33

4 Related Work **35**
4.1 Semantic Formalisations of LLVM IR 35
4.1.1 Vellvm and VIR 35
4.1.2 K-LLVM 36
4.2 Verification Tools Targeting LLVM IR 36
4.2.1 LLVM Model Checkers 36
4.2.1.1 LLMC 36
4.2.1.2 RCMC 37
4.2.2 Other (Bounded) Verifiers for LLVM IR 37
4.2.2.1 Serval 37

4.2.2.2 FauST . . . 37
4.2.2.3 SAW . . . 38
4.3 High-level LLVM IR Abstractions . . . 38
4.3.1 LLVM C Backend . . . 38
4.3.2 SMACK . . . 38
4.3.3 LLVMVF . . . 39
4.3.4 SeaHorn . . . 39
4.4 Control Flow Structuring . . . 39
4.4.1 Control Flow Graph Reducibility . . . 40
4.4.2 Existing Decompilers . . . 41
4.4.2.1 Phoenix . . . 41
4.4.2.2 DREAM . . . 42
4.4.3 LLVM Loop Analysis Tools . . . 43

5 Tool Design and Architecture **45**
5.1 Embedding vs Externalising . . . 46
5.1.1 Embedding VCLLVM . . . 46
5.1.1.1 GraalVM and SuLong . . . 47
5.1.1.2 Assembly Languages as Verification Target . . . 48
5.1.1.3 JavaCPP . . . 48
5.1.2 Externalising VCLLVM . . . 49
5.1.2.1 Language Choice . . . 50
5.2 VCLLVM Output Format . . . 50
5.2.1 Using a Concrete Syntax . . . 50
5.2.2 Serialising COL . . . 51
5.2.2.1 COL JNI Bindings . . . 51
5.2.2.2 COL Protobuffers . . . 52
5.3 Specification Syntax Design . . . 52
5.4 Design Summary . . . 55

6 Tool Implementation **57**
6.1 Analysis Pass Hierarchy . . . 57
6.2 Specification Parsing . . . 58
6.3 The Origin System . . . 61
6.3.1 The `prefferedName` Origin Field . . . 61
6.3.2 The (`inline`)`Context` Origin Field . . . 61
6.3.3 The `shortPosition` Origin Field . . . 62
6.4 Regression Testing . . . 63

7 Usage & Examples **67**
7.1 Tool Usage . . . 67
7.2 Verification Examples . . . 68
7.2.1 Triangular Numbers and Cantor Pairs . . . 69
7.2.2 Date Comparison . . . 72
7.2.3 Fibonacci Sequence . . . 76

8 Conclusion **79**

9 Future Work **83**
9.1 VCLLVM Feature Road Map . . . 83
9.2 Abstracting Low-level and LLVM-specific Constructs . . . 85
9.2.1 Type Semantics . . . 85
9.2.1.1 Integers . . . 85
9.2.1.2 Pointers . . . 86
9.2.1.3 `undef` and `poison` Types . . . 87
9.2.2 Control Flow Structuring . . . 87
9.3 Extending Test Infrastructure . . . 88
9.4 External Library Support . . . 89
9.5 Memory Model . . . 89
9.5.1 Memory Layout . . . 89
9.5.2 Permissions . . . 90
9.5.3 Memory Atomics . . . 91

A Specification Expression Grammar **103**

B Additional Verification Examples **105**
B.1 Types . . . 106
B.1.1 Integers . . . 106
B.1.2 Booleans . . . 109
B.2 Arithmetics . . . 110
B.2.1 Addition, Subtraction, and Multiplication . . . 110
B.2.2 (Un)signed Division . . . 111
B.3 Branching . . . 113
B.3.1 If-then . . . 113
B.3.2 If-then-else . . . 114
B.3.3 Nested Branches . . . 116
B.4 Functions . . . 118
B.4.1 Function Call . . . 118
B.4.2 Cyclic Function Call . . . 118
B.4.3 Multiplication with Recursive Summing . . . 119
B.4.4 Factorial . . . 120

Chapter 1

Introduction

Errors in computer programs are as old as the discipline of computer programming itself. Ever since the first computers were built, humanity has been in a fight against these errors whether they are caused by human beings or anomalies in the system.

Errors in computers or the program it runs are colloquially known as bugs. The origin of the term is unclear and even dates as far back as the 19th century when Thomas Edison wrote the following in a letter to one of his associates [21] when talking about the process of inventions:

> *The first step is an intuition—and comes with a burst, then difficulties arise. This thing gives out and then that–* ***"Bugs,"*** *as such little faults and difficulties are called, show themselves.*

The first-ever (and quite literal) bug report was submitted by Grace Hopper in the 1940s concerning an actual bug getting stuck in one of the vacuum tubes of the Mark II [16]. The bug report can be found in Figure 1.1.

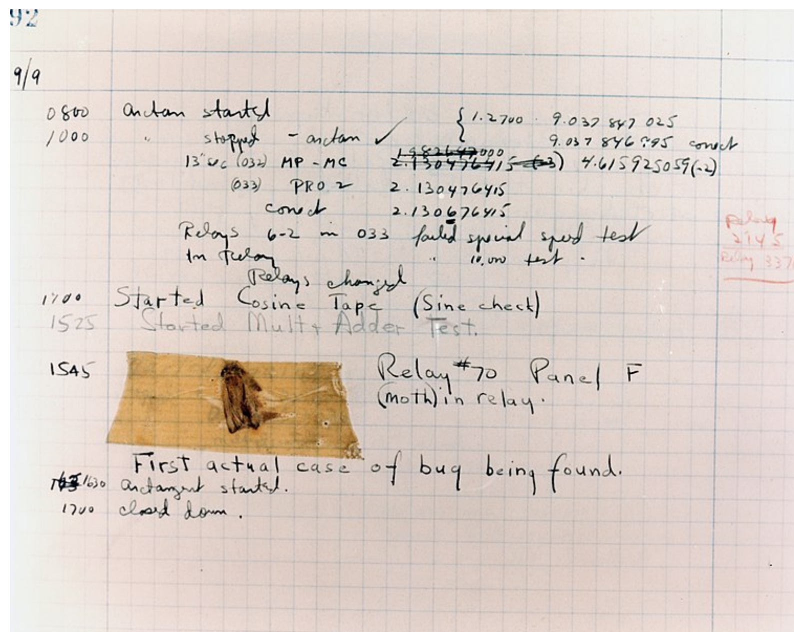


FIGURE 1.1: The first-ever bug report. Retrieved from the Wikimedia Foundation[1].

[1]See: `https://commons.wikimedia.org/wiki/File:First_Computer_Bug,_1945.jpg`

Ever since the invention of the transistor, technology has moved away from these "buggy" vacuum tube computers in favour of silicon-based hardware. Although safer from non-human errors, in rare cases, external hardware bugs can still occur. Cosmic background radiation can wreak havoc on the most advanced computer systems[2].

However, at some point, it is time to stop blaming external factors and look inwards and accept the fact that most bugs are caused by human mistakes. For example, studies in the 80s and late 90s agreed that anywhere from 1 to 25 bugs occur per 1000 lines of code in software that runs in production [6, 23, 29, 30, 42, 74]. While a more recent study suggests a much lower number of 1 bug per 1600-2500 lines of code [31], it still goes to show that even the best computer programmers in the world make mistakes which makes the practice of software verification so important.

There are many different software verification techniques that all provide different degrees of certainty about the correctness of a system. For example, systems can be dynamically verified (i.e. executed and observed) using various testing techniques such as unit testing, integration testing, system testing and many more. More thorough verification often takes the form of static verification where instead of executing a program and observing the results (as is the case with dynamic verification), proofs on properties of programs are derived using mathematical models.

One such approach is model checking. There are lots of different kinds of model checking but the general idea is to take a program and abstract it to a *finite state machine*. The state machine can in turn be exhaustively searched based on specific properties the program should have.

Even stronger (i.e. more thorough) methods of static verification are often logic based. These logic-based proves can either be performed with pen and paper, partially assisted with a tool (often referred to as interactive theorem provers), or in an auto-active manner where the user needs to insert the right annotations with verification specifications and a tool will generate the proof based on said annotations.

One such auto-active verification method is deductive verification. Deductive verification works by annotating code with mathematical expressions describing the intended behaviour of a particular piece of code. These annotations are more colloquially known as specification contracts and deductive verifiers attempt to automatically prove these specifications alongside a semantic model of the code they are associated with.

At the Formal Methods and Tools group at the University of Twente, such a deductive verifier called VerCors (Section 2.1) is in active development. It supports a multitude of languages including Java, C (with OpenMP extensions), and PVL (Prototypal Verification Language) which is an internal language used to demonstrate and test features of VerCors. Internally, the verification engine uses Permission-Based Separation Logic (PBSL) (Section 2.1.2.2) which is a variant of Hoare logic (Section 2.1.2.1) with some extensions to also be able to reason about concurrent programs.

[2] The physics-based educational YouTube channel "Veritasium" made an entertaining and well-researched video about the topic: `https://youtu.be/AaZ_RSt0KP8`

A current issue for VerCors among other deductive verification tools is language support development and maintenance. A significant part of the development of a deductive verifier is defining language semantics. The trouble does not stop there however as the syntax and semantics of modern languages are also being extended and changed over time, creating an ever-lasting maintenance burden.

Enter the LLVM (Low-Level Virtual Machine) Project (Section 2.2) whose inception was inspired by a similar problem except in language compiler technology. The LLVM project recognised that it is a cumbersome endeavour to develop and maintain compilers for multiple CPU architectures for multiple programming languages.

The proposed solution is to have a universal intermediate representation for both source languages and instruction sets of CPU architectures. This representation was aptly named LLVM IR (with IR standing for intermediate representation) (Section 2.2.1). LLVM IR is a mix of both the common elements of many source languages (e.g. functions, a type system, and support for custom data structures) as well as the common elements of many assembly languages (e.g. a load/store architecture and being based on a register machine).

Since the LLVM Project has solved many development overhead issues for language compiler developers, it raises the question of whether it can achieve the same for VerCors.

This thesis presents VCLLVM (VerCors Low-Level Virtual Machine), a possible solution for the language support development overhead currently present in VerCors. Its goal is to add LLVM IR support to VerCors. With VCLLVM, it should in theory be possible to compile any source language that has an LLVM compiler to LLVM IR and verify the resulting program with VerCors. To the best of our knowledge, the tandem VCLLVM/VerCors presents a novel development in static verification research as it would be the first unbounded deductive verifier for LLVM IR.

However, there are significant challenges that VCLLVM has and still needs to overcome to be successful. The central research question and several challenges are discussed in Chapter 3.

VCLLVM has been inspired by other bodies of work concerning the LLVM Project and LLVM IR, and other low-level code representations (Chapter 4). Several research projects have developed formal semantics for LLVM IR (Section 4.1). There is also a plethora of verification tools targeting LLVM IR including model checkers (Section 4.2.1) and other bounded verifiers (Section 4.2.2). Additionally, there have been several projects that have written backend compilers for LLVM that target high-level languages and other intermediate representations that are commonly used for verification purposes (Section 4.3). As it turns out, control flow structuring is essential for VCLLVM to function better in the future on which plenty of research has been performed as well (Section 4.4).

With all the related work in mind, the design of VCLLVM is discussed in Chapter 5. In this section, different design choices are discussed such as embedding VCLLVM inside VerCors versus developing VCLLVM as an external tool (Section 1), the desired output format of VCLLVM (Section 5.2), and the syntax design of the specification language (Section 5.3).

Interesting implementation details are discussed in Chapter 6. These include the analysis pass hierarchy of VCLLVM (Section 6.1), the specification parsing implementation (Section 6.2), the implementation of the origin (i.e. the origin of errors in code) system (Section 6.3), and the regression testing suite of VCLLVM itself (Section 6.4).

How to use VCLLVM/VerCors is discussed in Section 7.1 including several examples in Chapter 7.

The thesis concludes by revisiting the Problem Statement in the Conclusion (Chapter 8). Finally, the thesis ends with an extensive future work section (Chapter 9) on what the future holds for VCLLVM and suggestions on how to improve the current implementation.

N.B. Because this thesis makes extensive use of domain-specific terminology, a glossary and acronym list have been included at the end of the thesis. Every initial use of a term listed in this glossary is underlined throughout the thesis.

Chapter 2

Background

This chapter covers the relevant background on VerCors and the LLVM Project in Sections 2.1 and 2.2 respectively.

2.1 VerCors

VerCors [8] is a verification tool developed by the Formal Methods and Tools group at the University of Twente. Its aim is to verify programs that support various forms of concurrency models statically. It supports several programming languages such as Java, and C with some extensions (for example OpenCL, and OpenMP). Additionally, VerCors features PVL which is a Java-like toy language used to prototype, test, and demonstrate verification features.

VerCors is a complex tool with several internal and external components which are described in Section 2.1.1. Verification happens through user-provided specifications that are written in the specification language of VerCors which is covered in Section 2.1.3. The specification language uses *permission-based separation logic* (PBSL) to reason about concurrency which is explained in more depth in Section 2.1.2.

2.1.1 VerCors Architecture

The VerCors toolset can be broken down into three main components as visualised in Figure 2.1. A program being verified passes these three components in the following order:

1. The **language parser** converts a program written in any of the supported languages into an abstract syntax tree (AST). This AST is written in the common object language (COL). COL is meant to be only used in the tool internally and therefore has no concrete syntax.

2. The **rewriter** which simplifies the obtained COL from step 1 through several compiler-like passes. It converts high-level language constructs such as classes and complex abstract data types into simpler, but semantically equivalent types. This is necessary since the eventual verification language used in step 3 only supports relatively simple language constructs as opposed to a language like Java that is relatively abstract.

3. The **verifier** performs the computationally intense part of the verification process to reach a verdict about the correctness of the program regarding its specifications. This happens by converting the simplified COL AST obtained from step 2 into a

program specification written in Viper. Viper [43] is a verification tool developed by the Programming Methodology Group of ETH Zürich. Viper also uses separation logic to deductively verify a given program against its specification.

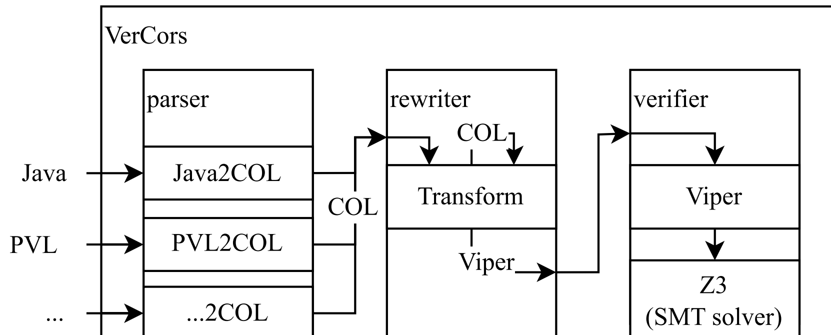


Figure 2.1: Architecture of the VerCors tool set.

2.1.2 Permission-Based Separation Logic

Both Viper and VerCors use permission-based separation logic (PBSL) to reason about concurrent programs. Before being able to prove anything about the functional correctness of a program, one needs to first verify there are no data races[1] inside the program to ensure consistent execution of the program. PBSL can prove this data race freedom. PBSL is an extension of *Hoare logic* which will be covered first.

2.1.2.1 Hoare Logic

Hoare logic [27, 47] provides a logical framework to reason about the (partial) correctness of programs. The core feature of Hoare logic is the *Hoare triplet* with the form of $\{P\}Q\{R\}$. Informally this notation means: "If conditional statement P holds before the execution of program Q, then conditional statement R holds after the execution of Q". Furthermore, P and R are referred to as precondition (a statement that should hold before executing Q) and postcondition (a statement that is guaranteed to hold after executing Q if the precondition before Q also held) of Q respectively.

Additionally, Hoare logic is compositional meaning:

$$\{P_1\}Q_1\{R_1\} \wedge \{R_1\}Q_2\{R_2\} \rightarrow \{P_1\}Q_1;Q_2\{R_2\} \tag{2.1}$$

This means proofs for small programs can be composed into proofs for larger programs.

[1] Data races are covered in more depth in Section 2.1.2.2.

To make the concept of Hoare logic more concrete, consider the following Java program fragment:

Listing 2.1: **Java** program fragment.

```
1 ...
2 public static int mult_add_2(int x) {
3     assert(x > 0);
4     x = x + 2;
5     int ret = x * 2;
6     return ret;
7 }
8 ...
```

It is a simple function that takes an input value of type `int`, adds 2, doubles this value, and finally returns the doubled value. The assertion on line 3 can be interpreted as a precondition $\{x > 0\}$. By applying the appropriate axioms for addition and assignment to line 4, the following Hoare triplet can be derived:

$$\{x > 0\}\texttt{x = x + 2 }\text{(line 4)}\{x > 2\} \tag{2.2}$$

If the postcondition of Equation 2.2 is used as a precondition for line 5 and combining the composition rule listed in Equation 2.1, and the Hoare axioms for multiplication and assignment, the following derivation can be made:

$$\begin{aligned}
&\{x > 0\}\texttt{x = x + 2 }\text{(line 4)}\{x > 2\} \\
&\wedge \\
&\{x > 2\}\texttt{int ret = x * 2 }\text{(line 5)}\{x > 2 \wedge ret = 2x\} \\
&\rightarrow \\
&\{x > 0\}\texttt{x = x + 2 }\text{(line 4); }\texttt{int ret = x * 2 }\text{(line 5)}\{x > 2 \wedge ret = 2x\}
\end{aligned} \tag{2.3}$$

By substituting $x > 2$ into $ret = 2x$, the postcondition $ret > 2(x + 2)$ holds. Thus Equation 2.3 provides the full proof for the function `mult_add_2` that if the precondition $x > 0$ is satisfied, the function is guaranteed to return a value larger than $2(x + 2)$ of type `int`.

So far, Hoare logic seems to be limited to sequential loop-free programs. In theory, to support loops, all loop iterations could be unrolled, but this would be a tedious process and only possible in a select few cases where the amount of loop iterations is fixed. Therefore, Hoare logic provides the theory to reason about loops using loop invariants. Loop invariants are logical statements that should hold just before entering a loop, after each loop iteration, and just after exiting a loop.

To make the concept of loop invariants more concrete, consider the following Java fragment:

Listing 2.2: **Java** program fragment.

```
1  ...
2  public static int mult(int x, int y) {
3      assert(y >= 0);
4      int ret = 0;
5      for (int i = 0; i < y; i++) {
6          ret = ret + x;
7      }
8      assert(ret == x * y);
9      return ret;
10 }
11 ...
```

It is a simple function that takes two arguments, `int x` and `int y`, and multiplies the two arguments by repeated addition. Note that this technique only works when the argument `int y` is non-negative, hence the assertion in line 3.

The assertion on line 3 could be considered the precondition for the program. Furthermore, the assertion on line 8 could be considered the postcondition of the program. To prove this program against its specification, it is required to reason about the semantics of the loop on line 5.

This example perfectly demonstrates the need for loop invariants in Hoare logic. Because there is no upper bound to the value of `int y`, there is a virtually infinite number of possible executions of the loop. This makes it impossible to prove the specification without a loop invariant as it is impossible to cover an infinite number of executions.

There are many loop invariants that hold for the loop of this program. For example, $i \geq 0$ or $i \leq y$ or even irrelevant expressions such as $5 = 5$. However, none of these will help the generation of the proof for the given pre- and postcondition. For this to be possible, a loop invariant is needed to describe and relate changes to variables `int i` with respect to `int x`, `int y`, and `int ret`, for example:

$$0 \leq i \leq y \wedge ret = x \cdot i \tag{2.4}$$

The proof techniques for proving loop invariants and the formal proof that this loop invariant holds are beyond the scope of this thesis. However, to provide some intuition on how loop invariants are used, an informal proof can be found in Listing 2.3.

Listing 2.3: **Java** program fragment from Listing 2.2 annotated.

```
1  ...
2  public static int mult(int x, int y) {
3      assert(y >= 0);
4      {y ≥ 0}
5      int ret = 0;
6      {y ≥ 0 ∧ ret = 0}
7      loop invariant: 0 ≥ i ∧ i ≤ y ∧ ret = x · i (Equation 2.4)
8      This holds at the beginning of the loop for i = 0 ∧ ret = 0 ∧ y ≥ 0
9      It will hold after each loop iteration because i will never exceed y and
10     because after i iterations, x has been added i times to ret i.e. ret = x · i
11     for (int i = 0; i < y; i++) {
12         ret = ret + x;
13     }
14     Since the loop is exited, by induction: i = y
15     ret = x · i as we are at the end of a loop iteration
16     Finally, ret = x · i ∧ y = i → ret = x · y □
17     assert(ret == x * y);
18     return ret;
19 }
20 ...
```

Lastly, it is important to note that to reason about a program using Hoare logic without any extensions, the program is required to be structured and reducible [41].

There are many definitions of *Structured programming* but in this thesis, the definition of Dahl, Dijkstra, and Hoare is adopted [15]. This definition is based on that every structured program is decomposable into three core structures which are concatenation (i.e. lines of codes being executed sequentially), selection (e.g. if-then-else structures), and repetition (e.g. loop constructs like for- and while-loops). Note that the absence of arbitrary goto statements is paramount to structured programming.

The reducibility property of programs is extensively discussed by Allen [1] and covered in more detail in Section 4.4.1.

2.1.2.2 Permission-Based Separation Logic

Although Hoare logic gives means to reason about sequential programs, it does not provide any methods to reason about the shared state between multiple processes. Due to the possibility of processes interleaving one another, it can be the case that a concurrent program has different possible outputs for the same input.

As an example, consider Figure 2.2. It describes the flow of a program initialising a variable x, forking into two separate threads that each increment x by one, and finally join to return the value of x. The value of x at the end of the program is dependent on the execution order of the individual steps of the program. For example, the order $\{a_1, a_2, a_3, b_1, b_2, b_3\}$ would produce $x = 2$, whereas $\{a_1, b_1, a_2, a_3, b_1, b_2, b_3\}$ would produce $x = 1$. In the second example, both threads will read $x = 0$, compute $tmp = 1$, and subsequently write $x = 1$.

These kinds of situations where two or more threads try to access the same memory location at the same time are known as *data races*.

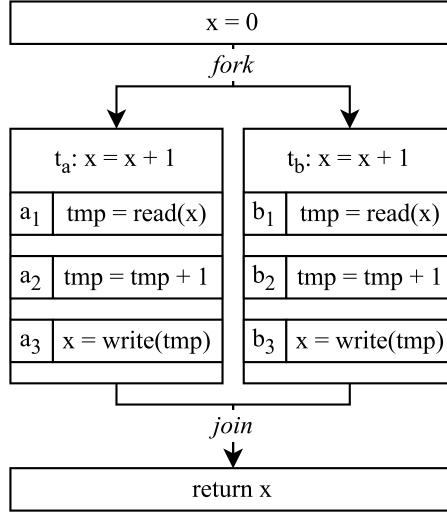


FIGURE 2.2: Example model of a program containing a data race.

Data races make it harder to reason about the semantics of a program as the number of possible outcomes grows exponentially with respect to both the number of threads and times a data race is triggered. It is even worse when in a parallel context, both writes happen at the same time, likely resulting in corrupted memory. Data races are therefore considered undesirable.

PBSL [9, 55] provides the means to prove the absence of data races. It introduces the notion of permissions or ownership over heap memory. Dijkstra [19] already observed that writes to a shared variable are always safe (read: do not involve data races) if that variable is only accessed in mutually exclusive sections of the program. Moreover, it was observed that concurrent reads to a shared variable by multiple threads are safe. After all, the value is not being altered so it is inconsequential in what order the threads read out the variable.

This notion has been formalised in PBSL by using fractional permissions. Every heap variable is assigned a permission value p. Threads can own fractions of this permission in a range of $0 \geq p \geq 1$. The sum of all fractions of a permission value should always be at most 1. If a thread has $p > 0$ over a variable, it has read permission over that variable. Furthermore, if $p = 1$, it also has write permission. Throughout the rest of this thesis, the following notation will be used to indicate permission:

$$Perm(x, n)$$

Where x is a heap variable and n is the associated permission fraction.

By reasoning about the transfer of ownership of permissions between threads, proofs can be constructed about the absence of data races. To further illustrate the basic concepts of PBSL consider the following Java fragment:

Listing 2.4: **Java** program fragment.

```
1  class IntCell {
2      int x = 0;
3
4      void increment() {
5          int tmp = this.x;
6          tmp++;
7          this.x = tmp;
8      }
9  }
10
11 class IncrementTask extends Thread {
12     IntCell cell;
13
14     IncrementTask(IntCell cell) {
15         this.cell = cell;
16     }
17
18     @Override
19     public void run() {
20         cell.increment();
21     }
22 }
```

Two classes are introduced to model the example in Figure 2.2.

The first class is the `IntCell` class. It is a simple holder class for an integer value x. For simplicity, proper encapsulation principles have been disregarded. The `increment` method closely models the behaviour of a singular thread as presented in Figure 2.2: It reads its internal integer value into a temporary variable, increments the temporary variable by one, and finally writes the temporary value back to its internal integer.

The second class is the `IncrementTask` class. Its only purpose is to be run in a separate programming thread and increment the value of a possibly shared `IntCell`.

Extrapolating the behaviour of the example in Figure 2.2 to this example, `IncrementTask` could cause data races in a concurrent context. For example, the following application of `IntCell` and `IncrementTask` would trigger a data race for $tCount > 1$:

Listing 2.5: Example usage of `IntCell` and `IncrementTask` (Listing 2.4) in **Java**.

```java
...
static int incrementConcurrently(int tCount) throws InterruptedException {
    assert(tCount >= 0);
    var cell = new IntCell();
    IncrementTask[] threads = new IncrementTask[tCount];
    for (int i = 0; i < tCount; i++) {
        threads[i] = new IncrementTask(cell);
        threads[i].start();
    }
    for (Thread t : threads) {
        t.join();
    }
    assert(cell.x == tCount);
    return cell.x;
}
...
```

This method is similar to what is presented in Figure 2.2 except for an arbitrary number of threads instead of just two presented in the diagram.

The assertion on line 3 could be interpreted as a precondition. After all, running a negative number of threads is impossible, but running none is a possibility.

Similarly, the assertion on line 13 could be interpreted as a postcondition. It is reasonable to expect that running an increment on $cell.x$, $tCount$ times would result $cell.x$ to equal $tCount$.

Formalising these assertions yields:

$$\begin{aligned} &\text{precondition: } tCount >= 0 \\ &\text{postcondition: } cell.x = tCount \end{aligned} \tag{2.5}$$

This is however empirically disprovable due to the presence of data races on the value of $cell.x$. Table 2.1 shows the result of 10000 sample runs each having $tCount = 100$. While producing the correct result the majority of the time, it does suffer from inconsistent behaviour caused by the data race.

Value of $cell.x$	**Frequency**
98	1
99	16
100	9983

Table 2.1: Summary of 10000 execution of `incrementConcurrently` (Listing 2.5) (OpenJDK 11.0.16/Linux 5.10.0-19/AMD Ryzen 5 PRO 5650U/DDR4 @3200MHz).

In terms of PBSL semantics, nothing is stopping multiple threads from acquiring read and write permission over $cell.x$ at the same time.

More formally, assume that two arbitrary `IncrementTasks` t_1, and t_2 share a single `IntCell` c in the context of the method `incrementConcurrently`. Moreover, assume both t_1, and t_2 are executing `cell.increment()` concurrently. At that point in time, for both t_1 and t_2, $Perm(cell.x, 1)$ holds. This would indicate that a *separating conjunction* (indicating the combination of two disjoint permissions) $Perm(cell.x, 1) * *Perm(cell.x, 1)$ would be possible. However, recombining the permissions $Perm(cell.x, p) = Perm(cell.x, 1 + 1)$ would indicate $p > 1$ which likely indicates the presence of a data race.

To prevent data races, access to an instance of `IntCell` needs to be limited to one thread at a time. This is commonly achieved by using a lock abstraction on concurrent critical parts of the program. Semantically, a lock is an object that is to be initialised in an open state. Furthermore, the code can require a thread to acquire a particular lock to proceed with its execution and force the thread to release the lock at a particular point as well. Lastly, a lock can only be acquired by one thread at a time.

Locks are therefore very useful to prevent parallel access to an object as threads must wait on their turn to acquire the lock when it is not available. An improved version of `IntCell` can be found in Listing 2.6. To make sure only one thread can access a given `IntCell` at the same time, an arbitrary lock object is initialised. Furthermore, threads need to *synchronise* on the lock object to proceed with its execution. It means that if any thread is inside a **`synchronized`** block of a particular object, no other thread can enter a **`synchronized`** block synchronising on that same object.

An intuition on how to prove the absence of data races using PBSL is included in the annotations of Listing 2.6.

LISTING 2.6: Improved **Java** implementation of `IntCell` introduced in Listing 2.4.

```
1  class IntCell {
2      int x = 0;
3      Object lock = new Object();
4
5      void increment() {
6          The critical concurrent part of the program
7          At this point Perm(this, 0)
8          synchronized (lock) {
9              The lock has been acquired, meaning Perm(this, 1)
10             int tmp = this.x;
11             tmp++;
12             this.x = tmp;
13         }
14         The lock has been released again, meaning Perm(this, 0)
15         This allows for other threads to acquire the lock and thus write permission
16     }
17 }
```

2.1.3 VerCors specification language

Hoare logic and PBSL have been embedded in the VerCors specification language. Since specification embedding into LLVM IR will be a large part of this research, it is important to cover the VerCors specification language and its features. Its core features have been outlined by Blom et al. [8][2].

The specification language is independent of the target language. This means that the specification language heavily relies on the target language syntax. In this section, the Prototypical Verification Language[3] (PVL) will be used alongside the specification language to explain the specification languages concepts. PVL is part of VerCors itself and is strictly used for verification purposes (i.e. it is not executable). It has features and syntax similar to Java.

2.1.3.1 Method Contracts

Although VerCors supports assertions, a more concise method to verify program behaviour is by using *method contracts*. Method contracts can be embedded in special comment blocks directly above the method definitions. In the case of PVL, they are inlined as normal code (i.e. living outside comment blocks). To demonstrate method contracts, consider the following PVL fragment:

Listing 2.7: **PVL** program fragment.

```
1  class Calculator {
2
3      requires x > 0;
4      ensures \result > y && \result == x + y;
5      pure int add(int x, int y) {
6          return x + y;
7      }
8
9      requires x > 0 && y >= 0;
10     ensures \result == x + 2 * y;
11     ensures \result == add(add(x, y), y);
12     pure int add_twice(int x, int y) {
13         return add(add(x, y), y);
14     }
15 }
```

It is a program with two functions: `add` that adds parameter `int x` and `int y` together and `add_twice` which does the same thing except add `int y` twice.

Both methods feature a method contract. It uses the keywords `requires` and `ensures` to define preconditions and postconditions respectively. The `\result` keyword is used in postconditions to refer to the return value of a method.

[2]For more up-to-date documentation on the specification language, see: `https://github.com/utwente-fmt/vercors/wiki/Specification-Syntax`

[3]For documentation on PVL, see: `https://github.com/utwente-fmt/vercors/wiki/Prototypal-Verification-Language`

VerCors does not only assume preconditions to be true, but it also checks whether they are being respected by the rest of the program. This is why `x > 0 && y >= 0` (line 9) is a precondition of `add_twice`. It ensures the calls to `add` inside `add_twice` respect the contract of `add`.

Finally, although both postconditions on line 10 and 11 are functionally equivalent, there is a subtle difference. The former uses arithmetic operations in its expression while the latter uses method calls of the program. Using method calls inside contracts is only allowed for `pure` methods which means that the method has no side effects. After all, the purpose of verification is to prove certain properties about a program, not change the state of the program. Removing the `pure` keyword from the `add` method would make the postcondition in line 11 fail.

2.1.3.2 Permissions

Permission reasoning was not required in Listing 2.7. This is because all variables were allocated on the program stack and therefore guaranteed to be thread local. However, this is different for the following PVL fragment:

Listing 2.8: **PVL** program fragment.

```
1  class Calculator {
2
3      int x;
4
5      context Perm(this.x, 1);
6      requires y > 0;
7      ensures this.x > \old(this.x) && this.x == \old(this.x) + y;
8      void add(int y) {
9          this.x = this.x + y;
10     }
11
12     context Perm(this.x, 1);
13     requires y > 0;
14     ensures this.x == \old(this.x) + 2 * y;
15     void add_twice(int y) {
16         add(y);
17         add(y);
18     }
19 }
```

This fragment is very similar to the one in Listing 2.7 with some key differences:

- Where `int x` used to be a parameter of the methods, it is now a field of the `Calculator` class. This creates a proof obligation on the usage of this variable. In this case, methods that use the heap variable `int x` need to give proof specifications in their contract about the usage of the heap variable for the prover to satisfy the created proof obligation.
- Since `add` and `add_twice` are now modifying the state of `Calculator`, the methods are not pure anymore and thus cannot be referenced in specifications.

- Because `int x` is now a field of `Calculator`, `int x` is now located in heap memory and requires permission reasoning (i.e. it could be that a `Calculator` object could be accessed by multiple threads).

- As the methods are now mutating the state of the object instead of returning values themselves, the `\result` keyword is unavailable now. Reasoning needs to occur on the state of the object instead of the functionality of the method. By using the `\old` keyword in postconditions, the state before the execution of the method can be referenced to compare the states of objects before and after the execution.

To reason about permissions in method contracts, the notation `Perm(var, f)` is used where `var` refers to a heap variable and `f` to a permission fraction. This behaviour is consistent with the mathematical definition described in Section 2.1.2.2.

Often with permissions, the method that requires certain permissions is expected to hand all of its held permissions back to the callee of the method. This makes it so that the preconditions and postconditions for permissions are often identical. VerCors has recognised this and therefore introduced the `context` keyword that is a shorthand to indicate that a certain expression is both a precondition and postcondition.

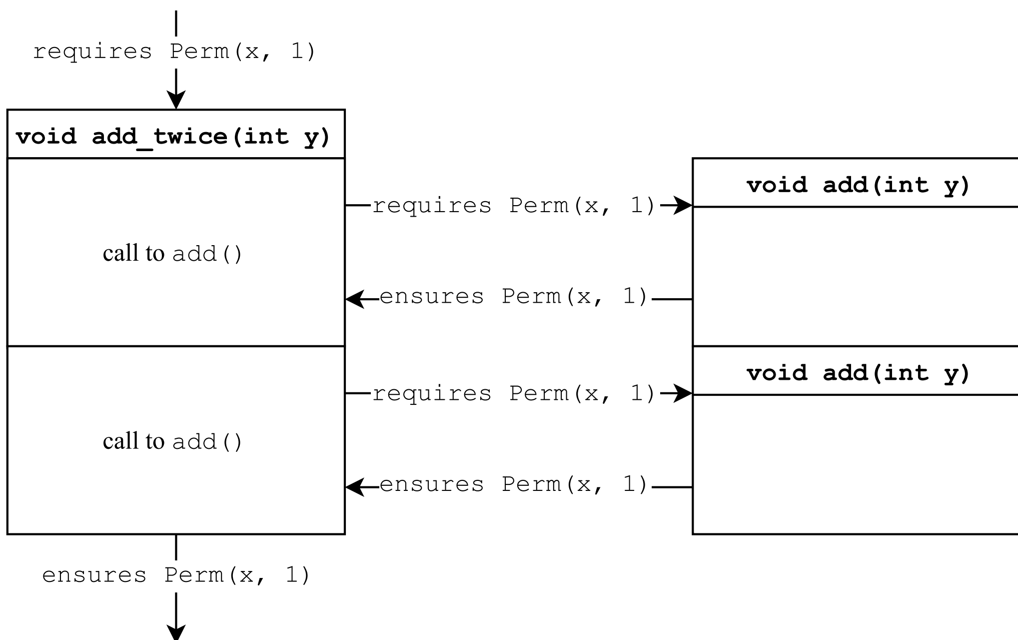


FIGURE 2.3: Permission flow of over the field `int x` for the method `add_twice`.

Figure 2.3 illustrates how pre- and postconditions regarding permission transfer should be interpreted. Pre- and postconditions involving permissions are analogous to transferring ownership to and from the method being called respectively.

It is important to be aware of permission leaks. That is when any permissions held by a method are not fully returned to the callee. A permission leak in the `add` method would result in a verification failure as the second call to `add` in `add_twice` would be unable to obtain the required write permission. This permission would have been lost after the first call to `add`.

However, permission leaks are not always undesirable. For example, intentional permission leaks could indicate that a certain variable becomes immutable after a certain action.

Although the program would technically still verify without having write permission over `int x` as a postcondition of `add_twice` (because `add_twice` is never called in the program), it is still undesirable because the `add_twice` method would be used in another method, the permission leak could potentially cause permission problems.

2.1.3.3 Arrays and Loops

Reasoning about loops and arrays comes with its own set of challenges. For example, it is simple to assert something about a single variable, but it gets more complex when reasoning about arrays that can take any size. Moreover, operations on arrays often require loops. It was established in Section 2.1.2.1 that to reason about loops in Hoare logic (and therefore by extension PBSL) loop invariants are required.

VerCors has features to support all these constructs, which will be explained through the following PVL fragment:

LISTING 2.9: **PVL** program fragment.

```
1  class ArrayLoop {
2      int[] input;
3      int[] out;
4      // permissions
5      context_everywhere Perm(input, 1\2);
6      context_everywhere input != null;
7      context_everywhere Perm(input[*], 1\2);
8      context_everywhere Perm(output, 1\2);
9      context_everywhere output != null;
10     context_everywhere Perm(output[*], 1);
11     // functionality
12     requires x > 0;
13     context_everywhere input.length == output.length;
14     ensures
15       (\forall int j; 0 <= j && j < input.length; output[j] > input[j]);
16     void map_add(int x) {
17         int i = 0;
18         // permissions
19         loop_invariant 0 <= i && i <= input.length;
20         // functionality
21         loop_invariant x > 0;
22         loop_invariant
23           (\forall int k; 0 <= k && k < i; output[k] > input[k]);
24         while(i < input.length) {
25             output[i] = input[i] + x;
26             i++;
27         }
28     }
29 }
```

It is a simple program that has an input and output array named `int[] input` and `int[] output` respectively. Additionally, it features a method `map_add` that takes the input array adds the parameter `int x` to each element of that array and writes the result to the output array. The following pre- and postconditions will be proven about the method `map_add`:

$$\begin{aligned}&\text{precondition: } x > 0 \land |input| = |output| \text{ (line 12)}\\&\text{postcondition: } \forall_{in_i \in input} \forall_{out_j \in output} (i = j \rightarrow out_j > in_i) \text{ (line 15)}\end{aligned} \tag{2.6}$$

The first new keyword that is introduced is `context_everywhere`. Unlike `context` that wraps a `requires` and `ensures` clause in one, `context_everywhere` goes a step further by additionally adding its expression to every loop inside the method body as a `loop_invariant`.

Acquiring permission over an array is slightly more complicated than acquiring permission over a single variable. The steps are as follows:

1. Permission needs to be acquired over the array data structure (line 5 and 8). This is because the array is located on the heap and thus contains a proof obligation regarding permission for its usage. For both `int[] input` and `int[] output`, only read permission is required. Only requiring read permission over `int[] output` may come as a bit of a surprise but since the data structure itself is not altered or replaced (merely its elements are edited), only read operations occur on the array as a whole.

2. A null check needs to occur on the array (line 6 and 9). Although it is more of a functional requirement than a permission-related requirement, it is essential to reason about the permission of elements within the array. Any reasoning about the elements of a `null` object cannot be performed.

3. Finally, permission can be acquired over the individual elements of the array (line 7 and 10). To reference elements of an array, the `array[*]` syntax can be used. Note that here write permission over `output[*]` is required because the values of the single elements are altered by the method.

The other interesting part of the fragment to discuss is the loop invariant. The loop invariant is almost completely devoid of any permission reasoning. This is because a lot of it is inherited from the `context_everywhere` clauses of the method contract. Regarding permissions, only line 19 is relevant. It ensures that `int i` will not exceed the length of the part of the input (and by extension the output (line 13)) that the method has permission over. In this case, it is the full length of the array. Without this line, the prover would not be able to verify whether the array bounds are exceeded by the loop and therefore breach permissions.

Lastly, the loop invariant in line 23 proves the methods postcondition (line 15) step by step. It is reminiscent of an inductive proof.

The loop invariant is a quantified expression. Quantified expressions can either be a `\forall` or an `\exists` matching their first-order logical counterpart ∀ and ∃ respectively. Quantified expressions have the syntactic form of *range iterator; range condition; Boolean expression*. For example, the line `\forall int i; i <= 0 && i < 10; array[i] == 0` would mean that the first 10 elements of `array` equal 0.

By relating `k` to `i` in a `\forall` expression, the proof of the postcondition is provided step by step:

1. At the beginning of the loop $i = 0$ and because $0 \leq k < i$ does not hold for any k, the loop invariant holds.

2. After each iteration, i is incremented and so does the range $0 \leq k < i$. Moreover, the expression `output[k] > input[k]` will hold for each new value of k as the new value of k equals $i - 1$ (the value of i of the previous iteration in which `output[i]` was set to `input[i]` increased by x (and $x > 0$ (line 21))).

3. At the end of the loop, the loop invariant will still hold as was the case at the end of every other loop iteration. Additionally, since $i = |input|$, the loop invariant is equivalent to the postcondition of the method (line 15) thereby proving the method functionality against its specification.

2.2 The LLVM Project

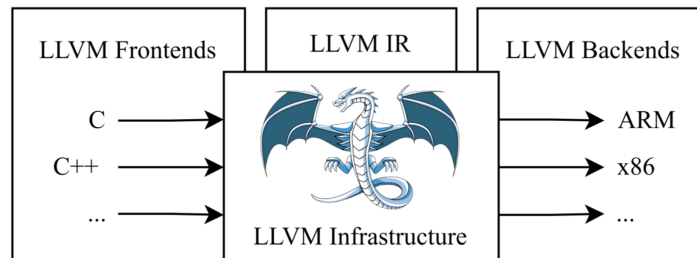


FIGURE 2.4: A simplified representation of the LLVM architecture.

LLVM (short for Low-Level Virtual Machine) is a project started in 2003 by C. Lattner and V. Adve at the University of Illinois [35]. Compilers are traditionally very specific tools for one source language to one binary target (for example, a C compiler for x86 or a Rust compiler for ARM).

The LLVM infrastructure unifies front- and backends as shown in Figure 2.4. LLVM aims to make all its frontends compile to a common intermediate representation and make all of its backends compile this common representation to a specific target. This common intermediate representation is the driving force of the LLVM infrastructure and is called LLVM IR. The premise of LLVM IR is to provide a common interface between both front- and backend compilers.

A frontend of a compiler provides the means to parse and transform a program written in a source language to LLVM IR, whereas a backend provides the means to transform LLVM IR into a target representation. The target representation of a compiler is often a binary or assembly format of a specific CPU architecture but can in theory be any form of data including another source language. LLVM therefore in theory also supports source-to-source transformations.

The obvious benefit of using the LLVM infrastructure is that frontend compiler developers only need to write a frontend for the LLVM infrastructure and do not need to worry about which targets to support. This similarly applies to the backend where supporting a target for the LLVM infrastructure unlocks a wide range of LLVM-based source languages that can run on the target.

The next Section (2.2.1) will dive deeper into LLVM IR as it provides the core on which LLVM operates and is the language this project aims to support for verification.

2.2.1 LLVM IR

LLVM IR (short for LLVM Intermediate Representation) is the internal representation language used in LLVM. It is designed to be abstract enough to be compiled from higher level frontend languages, but simple enough to be transformed into assembly or machine code for a specific CPU architecture. It is also the language being operated on by middle-end code optimisation and analysis passes [53]. This section is heavily based on the specification of LLVM IR as described in the LLVM IR documentation [51].

2.2.1.1 Representation Formats

LLVM IR itself comes in three equivalent formats:

1. As **in-memory compiler representation**. Used primarily to programmatically interact with the program code through the C++ LLVM API.
2. As **on-disk bitcode**. Used primarily for JIT compilation as it is optimised for fast memory loading.
3. As **human-readable assembly language**. Used primarily for debugging purposes.

Additionally, LLVM IR is a Single Static Assignment (SSA) representation [58]. SSA representations have a few characteristics which include:

- Each variable is assigned exactly once. Variables can optionally have several versions, usually indicated with a subscript number in the variable name.
- Conditional branching of an SSA program can be represented in a Control Flow Graph (CFG). The graph consists of blocks which are sequential pieces of code until a branch or jump occurs (indicated by (conditional) edges between blocks).
- On places where blocks converge in the CFG, Φ nodes are used to converge diverging versions of variables. More concretely, say blocks b_1 and b_2 assign x_1 and x_2 respectively and converge in b_3. Then in b_3, both outcomes can be summarised as $x_3 \leftarrow \Phi(x_1, x_2)$ for subsequent use of variable x.
- When a block b_0 branches out into n other blocks named $\forall_{j \in 1..n} b_j$, then b_0 *strictly dominates* (i.e. unreachable without passing through b_0 first) $\forall_{j \in 1..n} b_j$.

These SSA properties are useful to perform efficient optimisations on LLVM IR. For example, SSA has proven to be particularly useful to detect unused variables [58].

Lastly, it is worth noting that LLVM IR is not stable, meaning there are no guarantees for compatibility between LLVM IR produced between different LLVM versions [50]. This can pose problems when the implementation of an external tool is depending on the exact specification of a specific version of LLVM IR. This issue is discussed in detail in Section 3.2.1.

2.2.1.2 Component Hierarchy

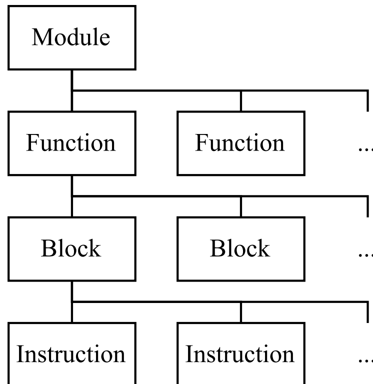


FIGURE 2.5: Hierarchy of the different LLVM IR components.

There are four levels to the component hierarchy of an LLVM IR program (Figure 2.5). Every component contains one or more of its child components in the hierarchy. For example, a module consists of one or more functions and a function consists of one or more blocks. The four LLVM components from the bottom up are:

1. An **Instruction** is the smallest unit in an LLVM IR program. Instructions can take several forms which are explained in more detail in Section 2.2.1.3.

2. A **Block** is an encapsulation of one or more instructions. They can be marked with a *label*. These labels can be referenced in branch instructions to jump to blocks within the same function. Additionally, the LLVM API provides tooling to extract CFGs of blocks on a function level. These can be useful to reason about and (re)construct loops and conditional logic of an LLVM IR program.

3. A **Function** is an encapsulation of one or more blocks. Each function has one block marked as *entry block* which is the block being executed when the function is called. Additionally, functions must have a return type (which can be void) and can have any number of arguments defined in their definition.

4. A **Module** is the highest level component in the hierarchy. Each LLVM IR file consists of one module and has no explicit syntax (i.e. there is no module declaration syntax, and a (source) file is implicitly compiled or parsed to a module). The module is an abstraction present in the API, rather than a syntactical object. For example, when loading an LLVM IR file into memory, a pointer to a module object is returned.

2.2.1.3 Features of LLVM IR

As mentioned in Section 2.2.1.2, LLVM IR is an SSA representation. The representation operates on a virtual register machine with an infinite number of virtual registers. The virtual register machine is a load/store architecture [35].

This section will highlight some of the features of LLVM IR and is mostly based on the documentation of version 15 of LLVM [51]. All sample code in this section has been compiled using either the `clang` or `clang++` compiler bundled with LLVM 15.0.6 using the `-emit-llvm` compiler flag. Parts of both source code and generated LLVM IR code may be omitted for readability.

Types LLVM IR features two basic types that are formally known as **first class types**. These are integers and floats. Both come in different flavours.

Integers can come in different width such as `i64`, `i32`, `i16`, and `i1` that could be used to map `long`, `int`, `short`, and `boolean` in an arbitrary source language respectively. LLVM makes no assumption whether the integer is signed or unsigned but will assume the integer to be written in two's complement if the integer is signed.

For floats, there is support for `half`, `float`, and `double` for different precision width floats. Floats correspond to the IEEE-754-2008 binary specifications. Additionally, it does support platform-specific float representations that do not adhere to IEEE-754-2008 such as `x86_fp80`.

These two types can be combined into aggregate types (for example vectors, arrays, and structs), or referenced via a pointer[4]. Pointers, arrays, and pointers themselves are considered first class as well. This allows for more complex nested data types as can be seen in Listing 2.10 and 2.11.

LISTING 2.10: Example of nested structs in **C**.

```
1  struct A {
2      int id;
3      char code[3];
4      char *data;
5  };
6
7  struct B {
8      double value;
9      struct A as[2];
10 };
```

LISTING 2.11: Structs in **LLVM IR** as generated by `clang`.

```
1  %struct.A = type {
2      i32,
3      [3 x i8],
4      ptr
5  }
6
7  %struct.B = type {
8      double,
9      [2 x %struct.A]
10 }
```

LLVM IR also supports type conversion operators from and to floats, integers, addresses, and pointers. There are also operators that can resize floats and integers to different widths.

Constants LLVM IR also supports constants. LLVM IR supports custom-declared constants and has a lot of predefined constants as well. For example, `true` and `false` can be used as Boolean constants and are of type `i1`. The constants `undef` and `poison` are particularly interesting because they can be used in composition with any first class type.

The constant `undef` is used to present undefined behaviour to the compiler. `undef` is useful as some language standards allow for undefined behaviour. `undef` represents a range of possible values and guarantees that no matter what value of `undef` is *chosen*, the program itself will remain well-defined (i.e. will not have erroneous behaviour). Note the subtlety of the word "*chosen*" in this context: if the `undef` value remains and is not optimised to a definite value, the behaviour of the program will not be well-defined.

[4]As of LLVM 15, opaque pointers (i.e. pointers without type) are preferred in favour of typed pointers (e.g. `i32*`) due to their better-defined semantics. Typed pointers are still supported in LLVM 15 but might become deprecated in subsequent versions. For more details on the topic, see: `https://releases.llvm.org/15.0.0/docs/OpaquePointers.html`

The constant `poison` on the other hand is used to indicate erroneous behaviour of a program (i.e. behaviour that is not well-defined). More specifically, `poison` behaviour violates assumptions made about the program. For example, poisonous behaviour could be a division by 0, an integer overflow, or an out-of-bounds index on an array. The presence of `poison` behaviour also very much depends on the constraints of the source language.

The difference between `undef` and `poison` can be better understood by example. Consider the three functions in both Listings 2.12 and 2.13.

LISTING 2.12: Three simple **C** functions.

```c
1  int undefX() {
2      int x;
3      return x;
4  }
5
6  int poisonX(int x) {
7      return x / 0;
8  }
9
10 int undefOrPoisonX() {
11     int x;
12     return 1 / x;
13 }
```

LISTING 2.13: Three **LLVM IR** functions (using `clang -O2`).

```llvm
1  define i32 @undefX() {
2    ret i32 undef
3  }
4
5
6  define i32 @poisonX(i32 %0) {
7    ret i32 poison
8  }
9
10 define i32 @undefOrPoisonX() {
11   ret i32 poison
12 }
13 ...
```

`undefX` indicates undefined behaviour but is not poisonous. That is, it is unclear what the value of `int x` should be, but no matter what value is *chosen*, the function produces well-defined behaviour (Note again here, that choosing a value is essential for the well-definedness property).

`poisonX` clearly presents poisonous behaviour. Namely, it is mathematically impossible to divide by 0. It presents erroneous behaviour and is therefore optimised to a `poison` value.

Lastly, `undefOrPoisonX` is a bit trickier to evaluate. At first glance, it seems the behaviour of the function is undefined as `int x` can take on any integer value. However, since the value of `int x` includes 0, not all values of `int x` present well-defined behaviour. Therefore, the behaviour of `undefOrPoisonX` is poisonous. In other words, `poison` values are stronger than `undef` values.

What values become `undef` and `poison` is largely dependent on which compiler flags, and which optimisation and (static) analysis passes are run by the compiler[5].

Binary Operators LLVM IR has support for common arithmetic binary operators on integers and floats or vectors thereof. For integer binary operations such as addition, subtraction, and multiplication, LLVM IR can use the same operator regardless of whether the integer is signed or unsigned as it assumes signed integers to be two's complement. The division of two's complement signed and unsigned integers have distinct computations.

[5]The optimisation passes the `clang` compiler can opt into are vast and is largely beyond the scope of this project. For the interested reader, see: `https://releases.llvm.org/15.0.0/tools/clang/docs/CommandGuide/clang.html`

Therefore, LLVM IR has distinct operators for signed and unsigned integers which are `sdiv` and `udiv` respectively. Listings 2.14 and 2.15 show how complex arithmetic C expressions get decomposed into several LLVM IR operations.

LISTING 2.14: Two similar **C** functions operating on integers and floats.

```c
int integerFunction(
    int a,
    int b,
    int c
    ) {
    return (a + b) * (a - c) / b;
}



float floatFunction(
    float a,
    float b,
    float c
    ) {
    return (a + b) * (a - c) / b;
}


...
```

LISTING 2.15: Two **LLVM IR** functions operating on integers and floats (using `clang -O2`).

```llvm
define i32 @integerFunction(
  i32 %0, i32 %1, i32 %2profit
  ) {
  %4 = add nsw i32 %1, %0
  %5 = sub nsw i32 %0, %2
  %6 = mul nsw i32 %5, %4
  %7 = sdiv i32 %6, %1
  ret i32 %7
}

define float @floatFunction(
  float %0, float %1, float %2
  ) {
  %4 = fadd float %0, %1
  %5 = fsub float %0, %2
  %6 = fmul float %4, %5
  %7 = fdiv float %6, %1
  ret float %7
}
...
```

LLVM IR also supports several bitwise binary operators for integers and vectors of integers. For most operators, it will assume the integer to be unsigned. There is one exception for `ashr` which performs a bitshift to the right meaning the sign is preserved. For example, shifting `ashr i8 -4, 2` would yield a value of -1.

Pointers and Aggregate Data Structures When writing values to stack memory, the process in LLVM IR is to first allocate space for the value in memory using the `alloca` instruction. This returns an address pointer to the value. This is then followed by a `store` instruction with the value and address pointer as arguments to commit the value to memory. Later, the value can be retrieved using a `load` instruction using the same address pointer.

This process is very straightforward when working with single value types but gets more complicated when working with aggregate data types. Because a memory allocation only returns one address pointer (to the beginning of the data structure), retrieving elements from the data structure requires pointer arithmetic.

Instead of letting compiler implementations handle address pointer arithmetic themselves, LLVM leverages its type system to do all necessary pointer calculations over aggregate data structures using the `getelementptr` instruction. The instruction performs a not-in-place calculation and returns a new address pointer.

The `getelementptr` instruction takes three or more arguments. The first argument is the aggregate data type over which the address pointer calculation should be done. The second argument is the actual address pointer to the data structure in memory which should match the type provided by the first argument.

The third and all subsequent arguments are indices that shift the address pointer around such that the new address pointer matches the location of the indices in the data structure in memory. For example, for a one-dimensional array one index argument should be provided, but for a two-dimensional array, two index arguments could be provided.

Lastly, there is an optional `inbounds` option that prevents the pointer from going out of the bounds of the data structure. When an invalid index is provided while using the `inbounds` option, a `poison` value will be returned.

To illustrate the utility of the `getelementptr` instruction, consider Listings 2.16 and 2.17. The example, albeit slightly complicated, comprehensively shows how `getelementptr` matches the behaviour of accessing an aggregate data structure with indices.

LISTING 2.16: Example of accessing an aggregate data structure through pointers in **C**.

```
1  struct S1 {
2      char A;
3      int B[10][20];
4      char C;
5  };
6  struct S2 {
7      int X;
8      double Y;
9      struct S1 Z;
10 };
11
12 int *getZ_B_5_13(struct S2 *s) {
13     return &s->Z.B[5][13];
14 }
15
16
17
18
19
20
21
22 ...
```

LISTING 2.17: Example usage of `getelementptr` in **LLVM IR** (using `clang -O2`).

```
1  %struct.S1 = type {
2      i8,
3      [10 x [20 x i32]],
4      i8
5  }
6  %struct.S2 = type {
7      i32,
8      double,
9      %struct.S1
10 }
11
12 define ptr @getZ_B_5_13(ptr %0) {
13   %2 = getelementptr inbounds
14     %struct.S2, address type
15     ptr %0, address value
16     i64 0, dereference pointer
17     i32 2, point to 3rd field S2.Z
18     i32 1, point to 2nd field S1.B
19     i64 5, point to B[5]
20     i64 13 point to B[5][13]
21   ret ptr %2
22 }
```

Program Control Flow To perform branch and loop behaviour, LLVM offers branch instructions that can conditionally jump to the beginning of any instruction block in the same function. There are several different branch instructions in LLVM but the simplest and most common one is the `br` instruction.

The `br` instruction either takes one block label argument for an unconditional jump or one conditional (`i1` value) followed by two block label arguments. The first and second labels indicate to which block the program should jump when the conditional is either true or false respectively.

Listing 2.18: **C** function that calculates positive factorials through repeated multiplication.

```
1  int factorial(int n) {
2      if (n < 1) {
3          return -1;
4      }
5      int i = n - 1;
6      while (i > 0) {
7
8
9
10
11
12
13
14         i--;
15         n *= i;
16
17     }
18
19     return n;
20
21
22
23
24 }
```

Listing 2.19: **LLVM IR** function that calculates positive factorials (using `clang -O1`).

```
1  define i32 @factorial(i32 %0) {
2    %2 = icmp slt i32 %0, 1
3    br i1 %2, label %11, label %3
4  3:
5    %4 = icmp eq i32 %0, 1
6    br i1 %4, label %11, label %5
7  5:
8    %6 = phi i32
9      [ %8, %5 ],
10     [ %0, %3 ]
11   %7 = phi i32
12     [ %9, %5 ],
13     [ %0, %3 ]
14   %8 = add nsw i32 %6, -1
15   %9 = mul nsw i32 %8, %7
16   %10 = icmp sgt i32 %6, 2
17   br i1 %10, label %5, label %11
18 11:
19   %12 = phi i32
20     [ -1, %1 ],
21     [ %0, %3 ],
22     [ %9, %5 ]
23   ret i32 %12
24 }
```

Branches enable LLVM to compile (among other constructs) if-then-else statements and loops. This is illustrated in Listings 2.18 and 2.19. The sample program can be broken down as follows:

- The if-statement conditional gets evaluated in the entry block of the program. From there it either branches to the last block in the function (block `11`) or to the next block (block `3`).

- Block `3` contains the condition of the while-loop. Note how the assignment of loop iterator `int i` has been completely moved into the loop body (block `5`). Block `3` basically determines whether we skip the loop (block `11`) or enter the loop body (block `5`).

- In Block `5`, the first Φ nodes are encountered. As mentioned in Section 2.2.1.1, Φ nodes are often necessary to merge two converging blocks together. This is the case for block `5` as it can be entered either after block `3` or block `5` itself. Moreover, both blocks define a value for `int n` stored in registers `%0` and `%8` respectively. Which register value should be picked is resolved with the instruction `%6 = phi i32 [ %8, %5 ], [ %0, %3 ]` (line 8).

- To use Φ nodes in LLVM IR, the `phi` operator should be used. The operator takes an arbitrary amount of register-block label pairs. The register should match the type of the Φ node. The pairs should be read one by one and can be interpreted as: "Take the value of the register (first pair argument) if the previously visited block label matches this block label (last pair argument)." So, in the example of line 8 from Listing 2.19 it means: "Take the value of register `%8` if the previously visited block is block `5` and take the value of register `%8` if the previously visited block is block `3`."

- Lastly, Block `11` marks the function exit. All return statements got squashed into a single block. Since there are three blocks that can potentially jump to this block which all define a separate value for `int n`, the block has a Φ node with three register-block label pairs.

Exceptions Although there is no definitive method to deal with error states among programming languages, LLVM provides a set of basic error handling instructions that when composed together can support a wide range of error handling paradigms. The mechanics of these instructions are discussed in more depth in the LLVM documentation [48]. With these instructions, error handling methods of various languages can be constructed.

Listing 2.20 gives a typical example of the C++ exception model. Listing 2.21 shows how this translates to LLVM IR. Note that the LLVM IR code is heavily simplified for readability.

The C++ example gets converted to LLVM IR as follows:

- Upon entry of the `try` block, memory needs to be allocated for exception information of exceptions that might get thrown. Through static analysis, LLVM can conclude that the only throwable is a value of `i32`, hence the allocation of 4 bytes in memory on line 5.

- In LLVM IR, methods that can potentially throw exceptions are `invoke`d rather than `call`ed. When an error is triggered, the call stack will unwind until it comes across an `invoke` instruction. The `invoke` instruction contains two labels. The first one indicates a jump when the function is returned and the second one indicates a jump when the function is being unwound.

- The actual throw function being invoked on line 7, takes three arguments: The exception object, the exceptions type info, and finally a destructor of the exception object. In this case, the destructor is `null` as the exception object has been allocated on the stack.

- At the point where the throw function has been invoked, the call stack will immediately get unwound and as can be read from line 11, the program jumps to block `2` coming across a `landingpad`. The `landingpad` on line 13 intends to catch a pointer to an `i32` value.

- The rest of block 2 further checks whether the exception caught by the `landingpad` is the actual exception it intended to catch. If it is, the program enters the `catch` block (block 7). Otherwise, it will continue unwinding to the next `invoke`/`landingpad` or exit the program entirely if the bottom of the call stack has been reached.

Listing 2.20: Two **C++** function showcasing the C++ exception model.

```cpp
1
2
3  int throwAndCatch() {
4      try {
5
6
7          throw -1;
8
9
10
11
12
13     } catch (int e) {
14
15
16
17
18
19
20
21
22         return e;
23
24
25
26
27
28
29     }
30
31
32
33 }
```

Listing 2.21: Two simplified **LLVM IR** equivalent functions (using `clang++ -O1`).

```llvm
1  @_ZTIi = external constant ptr
2
3  define i32 @throwAndCatch() {
4    %1 = call ptr
5      @allocate_exception(i64 4)
6    store i32 -1, ptr %1, align 16
7    invoke void @throw(
8      ptr %1,
9      ptr @_ZTIi,
10     ptr null
11   ) to label %12 unwind label %2
12 2:
13   %3 = landingpad { ptr, i32 }
14           catch ptr @_ZTIi
15   %4 = extractvalue
16     { ptr, i32 } %3, 1
17   %5 = call i32
18     @llvm.eh.typeid.for(ptr @_ZTIi)
19   %6 = icmp eq i32 %4, %5
20   br i1 %6, label %7, label %11
21 7:
22   %8 = extractvalue
23     { ptr, i32 } %3, 0
24   %9 = call ptr
25     @begin_catch(ptr %8)
26   %10 = load i32, ptr %9, align 4
27   tail call void @end_catch()
28   ret i32 %10
29 11:
30   resume { ptr, i32 } %3
31 12:
32   unreachable
33 }
```

Concurrency LLVM IR does not offer any functionality to create parallel threads or register signal handlers. However, it does support several instructions and flags that have well-defined behaviour in the presence of threads and other asynchronous programming constructs. The behaviour of atomic memory operations and ordering flags have been carefully described in the LLVM documentation [49, 51].

Apart from `store atomic` and `load atomic`, there are three other atomic (i.e. guaranteed to be executed without any interleaving) memory instructions, these are:

- The `cmpxchg` instruction loads a value from memory and compares the loaded value to a test value. If the loaded value and test value are equal, it overwrites the loaded value in memory with a given new value.
- The `atomicrmw` instruction modifies a memory location. However, unlike a `store` instruction that completely overwrites a previous value, `atomicrmw` additionally takes a binary operator (either bitwise or arithmetic) as an argument. The binary operator is applied to the provided value and the value in memory to create a new value in memory.
- The `fence` instruction does not modify or read any memory, rather it indicates to the compiler that all instructions before the fence, must be performed before any instructions after the fence.

LLVM	**C**	**Java**
NotAtomic	heap variable	-
Unordered	-	heap variable
Monotonic	`memory_order_relaxed`	-
Acquire	`memory_order_acquire`/ `memory_order_consume`	-
Release	`memory_order_release`	-
AcquireRelease	`memory_order_acq_rel`	-
SequentiallyConsistent	`memory_order_seq_cst`	`volatile`

TABLE 2.2: LLVM IR, C, and Java atomic ordering standards and their relation.

CPUs and compilers tend to reorder memory operations for optimisation. In single-threaded programs, this can be performed safely, but in a concurrent context, it can result in unpredictable behaviour.

Therefore, all atomic instructions take an ordering flag in LLVM IR. They are inspired and largely equivalent with atomic ordering rules set out by the ISO C11 standard [28] as well as the Java SE 8 language specifications [22]. The precise relation between the three standards is laid out in Table 2.2.

Additionally, Chakraborty and Vafeiadis [11] formalised the concurrency semantics of LLVM IR by using *event structures* [4]. Event structures can model LLVM IR Concurrency semantics accurately and can also be visualised in graphs. The contents of this work will however not be discussed here.

Chapter 3

Problem Statement

This chapter discusses the problem VCLLVM is attempting to solve. More specifically, the research question that this thesis answers is presented (Section 3.1), followed by the main challenges that were faced to answer the research question.

3.1 Research Question

The central research question of the VCLLVM project is:

What are the necessary steps to make the VerCors Toolset support and verify LLVM IR programs?

This question together with the background on VerCors (Section 2.1) and LLVM (Section 2.2) pose challenges that were tackled in the project to answer this research question.

3.2 Challenges

This section merely describes the challenges the project faced. Related research on how to tackle these challenges is discussed in Chapter 4. Actual design decisions related to the challenges are discussed in Chapter 5. Implementation details of how certain challenges were tackled are discussed in Chapter 6. Challenges that are still unresolved or only partially tackled are discussed further in Future Work (Chapter 9).

3.2.1 Challenge 1: Instability of LLVM IR

As mentioned in Section 2.2.1.1, LLVM IR is an unstable language [50]. Unstable means that LLVM versions are not backwards compatible. There is no guaranteed interoperability between the syntax of LLVM IR of different LLVM versions. What might be a valid LLVM IR program in one version of LLVM might not even be parsable by a few versions down the line.

It is desirable to always support the latest version of LLVM in VerCors. It can be challenging or even infeasible to stay up to date with the quarterly release schedule of LLVM if the right design choices are not made.

This issue is significant to keep in mind throughout the entire design process of VCLLVM but is most critical and discussed in detail in the design discussed in Section 5.1.

3.2.2 Challenge 2: Parsing Verification Specifications

As mentioned in Section 2.1.3, the VerCors specification language is very minimal and is supposed to be completed with the expressions from the source language.

However, this poses an issue as LLVM IR on a single instruction level is not very expressive. Instructions need to be sequenced together into blocks to achieve any kind of meaningful expression. Writing blocks in specifications or even multiple blocks (as for example would be the case for most Boolean expressions) would be impractical and above all, error-prone.

It should also be mentioned that the LLVM IR being written in SSA form is an upside regarding specifications. Since the reuse of variables is forbidden under the language constructs, it is hard to write specifications that have side effects. Except for memory instructions such as `store` and `alloca`, all instructions in LLVM IR are pure.

Design choices regarding the syntax of the specifications are discussed in Section 5.3. Implementation details on how specifications are parsed are discussed in Section 6.2.

3.2.3 Challenge 3: Origin of User Errors

When parsing an LLVM IR file with the parser of the LLVM Project, it either returns a module object or a diagnostics object containing parsing errors. While the diagnostics object does retain line and column numbers of the parsed file of where parsing errors occurred, the module object does not retain any origin information. The module object is merely a semantically equivalent in-memory representation of the program.

This naturally causes an issue in communicating the origin of an error in the original source code to the user. LLVM does offer the possibility of constructing a string of LLVM IR representing any LLVM value, but calculating line and column numbers or extracting an original source string is impossible as extraneous white spaces and comments in the original file are not persistent.

How VCLLVM deals with this challenge of communicating the origins of errors to the user is discussed in Section 6.3.

3.2.4 Challenge 4: VerCors and Compiler Intermediate Representations

Currently, VerCors only supports syntactically high-level languages such as C, Java, and PVL. This means the parsing and transformation process of these languages entails simplifying the program down to a level that is understandable for the verifier.

On the contrary, LLVM IR is essentially too basic for VerCors to understand. Higher level concepts, such as control flow structures (Challenge 5), will have to be reintroduced as part of the transformation process. Since this has not been done for any language supported in VerCors before, it might introduce implementation challenges in VerCors.

Moreover, LLVM IR introduces new paradigms never seen by VerCors before. These include a store/load architecture, register machines, and LLVM IR being an SSA representation. These new paradigms come with semantical concepts not yet supported by COL, such as loads, stores and other low-level memory instructions, Φ nodes, and low-level exception handling systems. Introducing all these new concepts to VerCors will likely pose some challenges in composing COL analogues or introducing new AST node types to COL.

The current iteration of VCLLVM simplifies many of these concepts or has not yet implemented them. Some ideas on how certain LLVM IR and low-level concepts could be translated into COL are discussed in Section 9.2.

3.2.5 Challenge 5: Control Flow Structuring

LLVM IR depends on jumps and branches (i.e. goto statements) in the function body to facilitate any control flow in a program, while VerCors requires structured, reducible programs to execute verification. VerCors has this requirement as its verification techniques are based on Hoare logic and as discussed in Section 2.1.2.1. This in turn requires structuredness and reducibility. VerCors does technically support goto statements but there are some caveats to be aware of when using them.

For example, the inclusion of goto statements obstructs the guarantee that the program is reducible [26]. Additionally, the need for structured programming becomes apparent when introducing loops to a program. Loop invariants become hard to verify when a loop is not structured (i.e. containing arbitrary goto statements).

With that in mind, from a compiler theory point of view, VCLLVM essentially needs to be an LLVM IR decompiler to the high-level COL language of VerCors.

Recovering high-level abstractions is a challenging endeavour when it concerns the control flow of a program. Loops can be especially hard to recover due to their various forms (e.g. for-loops, while-loops, and do-while loops). Moreover, different forms can be nested inside one another to introduce extra complexity.

The challenge arises not so much in detecting cycles in the CFG of the program (for which trivial graph algorithms exist), but once a cycle is detected, identifying the different parts of the loops. More specifically, identifying the loop condition, the loop body, and loop breaks can be challenging to identify. Additionally, nested branch structures come with their own set of challenges.

Control flow analysis of assembly-like programs is an extensively researched field and is discussed in more detail in Section 4.4.

3.2.6 Challenge 6: LLVM Concurrency Model

As discussed in Section 2.2.1.3, while LLVM IR does support instructions and control mechanisms that can be useful to ensure thread safety, it does not support constructs to deal with parallel thread creation or signal handling. LLVM IR code depends on being linked against existing concurrency libraries (e.g. the pthread library on POSIX systems for `clang`) to support this.

VerCors is a tool specialising in the verification of concurrent programs, so it is certainly enticing to support some kind of demonstrable concurrent features of LLVM IR. The choice of which concurrency libraries to support as well as defining their semantics can certainly pose a challenge.

This challenge is considered future work and briefly touched upon in Section 9.4.

3.2.7 Challenge 7: `undef` and `poison` Types

Both `undef` and `poison` types (Section 2.2.1.3) are semantically complex types. `undef` types are complex as they represent a set of possible values but should be semantically treated as if it is a single value. Moreover, what the value of `undef` variable ends up being on runtime is also completely dependent on the compiler backend.

`poison` types are complicated as they indicate erroneous behaviour. Poison values are however simpler to reason about semantically as poison values can always be treated the same. Whereas `undef` represents a (partially) unknown value, `poison` is more comparable to values such as `null`, or `NaN` and communicates a broken state of the program.

In any case, it will be challenging to capture the semantics of `undef` and `poison` into VerCors. While `poison` might be able to find a base in the VerCors exception handling model, `undef` seems to be a completely foreign concept in VerCors.

Ideas on how to deal with `undef` and `poison` is considered future work and briefly touched upon in Section 9.2.1.3.

Chapter 4

Related Work

To explore different angles to tackle the challenges in Section 3.2, this chapter discusses other research related to LLVM IR. It specifically focuses on work around the formal semantics of LLVM IR (Section 4.1), other verification tools targeting LLVM (Section 4.2), and more specifically tools that compile LLVM IR to higher level (verification) languages (Section 4.3). Finally, some research is discussed related to control flow structuring (Section 4.4).

Note that LLVM program verification is a vast research field. This means that although the section covers related research extensively, it is by no means complete.

4.1 Semantic Formalisations of LLVM IR

4.1.1 Vellvm and VIR

Besides the LLVM Language Reference [51] including a paragraph for each instruction explaining its semantics, several efforts have been made to formalise LLVM IR semantics.

One of these efforts has been the Vellvm framework [77]. It has formalised the semantics of LLVM IR inside the interactive theorem prover Coq. On top of that, they developed an interpreter for the semantic model such that Vellvm can be directly validated against a reference LLVM backend compiler implementation.

The VIR language [75] directly expands on Vellvm and mainly focuses on further validation of the semantics as well as dealing with the undefined behaviour of LLVM IR. Extensive validation of the VIR interpreter uses automatic unit test generation through HELIX [76]. To deal with undefined behaviour in LLVM IR, VIR uses *interaction trees* [72] that support possible divergent computations.

As it currently stands, the Vellvm framework has adopted VIR and currently supports LLVM 11[1].

While inspiration from the semantic language model from Vellvm and VIR can certainly offer inspiration for the implementation of LLVM IR in VerCors, there are some important issues to keep in mind.

Firstly, the semantic models are meant to be executable and are not intended for deductive verification. Yannick et al. [75] do however hint at how interaction trees could be refined down to Hoare logic.

[1] See: `https://github.com/vellvm/vellvm`

Secondly, as a side consequence of the executable nature of these semantic models, the semantics are very focused on modelling the memory state and layout of an LLVM IR program. This is hardly relevant to VerCors as it uses an implicit memory model rather than an explicit memory model.

4.1.2 K-LLVM

Vellvm and VIR can only semantically reason about sequential programs and fall short of reasoning about concurrent and parallel programs. K-LLVM [39] acknowledged this issue and developed a more complete semantic model of LLVM IR. Like Vellvm and VIR, it offers an executable semantic model but it is more complete by supporting the LLVM concurrency model.

As mentioned in Section 3.2.6, LLVM heavily relies on external library linking to support parallel thread creation and signal handling. K-LLVM chose to incorporate semantics for the POSIX pthread library (a popular system library for creating concurrent threads supported by most Unix-like operating systems) to support parallel thread creation to some capacity.

4.2 Verification Tools Targeting LLVM IR

4.2.1 LLVM Model Checkers

Model checking [12] is a verification technique where a large part or even an entire system is abstracted to a model, often some variant of a state machine. By using an appropriate model checking algorithm for the abstracted model, certain properties of the system can be exhaustively proven.
Several model checking tools have been developed around the LLVM Project. Although model checkers are fundamentally different from theorem-based proving tools such as VerCors, their approach to LLVM IR semantics could still be useful in understanding the expected behaviour of LLVM IR programs.

4.2.1.1 LLMC

LLMC [69] is an unbounded model checker for LLVM IR. Its main ability is to evaluate assertions in concurrent programs. Normally, assertions are difficult to test under concurrent systems as there is no guarantee all possible execution traces of the program are covered. LLMC can give stronger guarantees on unit tests due to evaluating every possible LLVM IR program trace through its model checker.

It supports the `__atomic_*` C/C++ built-ins, the POSIX pthread library, and the C standard library on top of LLVM IR itself. It is only able to reason about sequentially consistent programs.

LLMC uses the DMC model checker [70]. It first parses and translates an LLVM IR file to a DMC model that is consumable by the DMC API. DMC then checks the provided model by starting at the initial program state and recursively expanding all possible next states and checking encountered assertions in the process.

4.2.1.2 RCMC

RCMC [32] is a bounded model checker for LLVM IR. It targets concurrent programs and checks them for any assertion violation and race conditions. It proposes a model checking algorithm that generates and enumerates all possible execution graphs of a concurrent program. The algorithms are implemented using the RC11 memory model [34].

Unlike LLMC (Section 4.2.1.1), it supports the C11 relaxed memory model (or monotonic as it is known in LLVM) and thus does not require the program to be sequentially consistent.

4.2.2 Other (Bounded) Verifiers for LLVM IR

4.2.2.1 Serval

Serval [44] is a framework for developing low-level automated bounded verifiers. It mainly targets assembly languages. Although LLVM IR is not strictly an assembly language, a verifier for LLVM IR is also presented.

Serval is built on top of Rosette [68] which is a similar framework to Serval except more general and not necessarily aimed at assembly languages. Rosette also comes with a specification language that Serval also uses to write specifications.

Moreover, verifiers written in Serval produce an interpretable intermediate representation. This is useful for testing the correctness of the defined semantics of a verifier against a reference implementation of the language (or the other way around: useful for detecting bugs in the reference implementation).

While Serval takes a very similar approach to verification as VerCors, it has limits that VerCors with VCLLVM seeks to overcome. Most notably, verifiers developed using Serval can only reason about bounded loops, nor do they have support for any concurrency models.

4.2.2.2 FauST

FauST [56] is a verification framework for LLVM IR and can be used for any programming language that has an LLVM compiler frontend. It formalised the LLVM IR semantics into separate encodings. The encoding used depends on the verification technique used. FauST supports three verification techniques which are formal verification, automatic debugging, and automatic test generation. In the context of VCLLVM, formal verification is the most interesting.

Formal verification in FauST is supported through property checking (using assertions) and functional equivalence checking (i.e. providing a reference implementation of a program as the specification for the verifier). In either case, FauST has the same limitations as Serval (Section 4.2.2.1) meaning it is also a bounded verification framework and has no concurrency support.

4.2.2.3 SAW

SAW [20] is a verification tool that can transform programs into a set of logical SAWCore (the internal modelling language of SAW) expressions for a variety of languages including LLVM IR and Java bytecode. SAWCore can be simplified and optimised to be later used as input to external solvers and provers.

The transformation process to SAWCore uses symbolic execution to get a complete symbolic state model of the entire program, instead of bounding the program like most model checkers. It is therefore more useful for proving general functional correctness rather than finding specific execution paths that lead to erroneous behaviour as individual execution traces are abstracted away in SAWCore.

SAW is in this sense very similar to VerCors as both value general unbounded completeness of proofs over the identification of specific erroneous behaviour. A key difference is that VerCors is focused on proving specifications while SAW is focused on proving semantical equivalence between two programs.

4.3 High-level LLVM IR Abstractions

VCLLVM is essentially intended as a transformation tool from LLVM IR to COL. As discussed in Section 3.2.4, the main caveat is that COL in VerCors is a much more expressive and higher level language than LLVM IR and transforming a low-level representation to a higher level language is nontrivial, neither has it ever been done in VerCors.

However, transforming LLVM IR to higher level abstractions has been researched in the past with successful results and will be presented in the rest of this section.

4.3.1 LLVM C Backend

From the inception of the LLVM Project till the release of LLVM 3.1 [66], the LLVM Project included a backend compiler that compiled LLVM IR to C code. It was discontinued due to lack of maintenance which caused it to become unusable for any nontrivial C program. Several projects have attempted to resurrect the C backend for newer LLVM versions, most notably the version maintained by JuliaHub[2]. However, this version also struggles to find active maintainers which seems to be a recurring issue around the C backend.

Nonetheless, the C backend shows it is possible to write an LLVM backend compiler that targets a more abstract language.

4.3.2 SMACK

SMACK [54] is yet another verification tool targeting LLVM IR. However, SMACK is particularly interesting as it takes a unique approach to its LLVM IR abstraction.

All verification tools discussed until now use an external tool to abstract LLVM IR code to some model. However, SMACK directly implemented an LLVM compiler backend to compile LLVM IR to the Boogie intermediate verification language. It is very similar to how the C backend (Section 4.3.1) directly compiles to C code.

Boogie [5, 18, 36] is an object-oriented intermediate verification language that can be used as input for various solvers and verification tools such as Z3 [17], Dafny [37], and

[2]See: `https://github.com/JuliaHubOSS/llvm-cbe`

Chalice [38]. It serves a similar purpose as Viper in VerCors (Section 2.1). Boogie has support for assertions, axioms, and program specification contracts.

Currently, SMACK only supports bounded verification but claims to be extensible enough to also support forms of unbounded verification such as deductive verification in the future due to compiling to the versatile Boogie language.

4.3.3 LLVMVF

LLVMVF [63] is another bounded model checker, especially targeting concurrent LLVM IR programs. Like most other verification tools discussed so far that target concurrent programs, it supports the POSIX pthread library to reason about thread creation and signal handling.

LLVMVF is different and interesting for the following reasons:

1. LLVMVF uses the LLVM Haskell API instead of the native C++ libraries.
2. LLVMVF is not strictly a backend compiler. It rather focuses heavily on the analysis tools provided by the LLVM Project to extract a sensible model.

The tool works by taking an LLVM IR program containing some user-defined assertions. It then performs LLVM compiler passes that strip and optimise the code from unused constructs that are irrelevant for the model checker and unrolls loops. The resulting LLVM IR code is then used to extract an abstract concurrency model that can be fed to the model checker that will generate a verdict.

4.3.4 SeaHorn

SeaHorn [24] is a verification tool both similar to SMACK and LLVMVF. It is similar to SMACK in that it uses the native LLVM C++ API for interactions with LLVM IR, but also similar to LLVMVF in that it is not strictly a backend compiler but an abstraction extractor. In the case of SeaHorn, it extracts Constrained Horn Clauses (CHCs).

SeaHorn abstracts LLVM IR code along with its assertions to CHCs. This makes SeaHorn flexible as there are various verification tools that support CHCs as input, such as Spacer [33] and IKOS [10]. The mathematical foundations of CHCs are well beyond the scope of this thesis and will therefore not be discussed.

4.4 Control Flow Structuring

As mentioned in Section 3.2.5, recovering control flow structures of assembly(-like) code is a complex problem, especially when this code is highly optimised. Fortunately, control flow structuring and more broadly decompilation algorithms are an extensively researched topic in the field of computer science.

This section discusses some of this research by first covering some early work regarding CFG reducibility as well as interval analysis on CFGs (Section 4.4.1). This is followed by discussing existing decompilers and in particular their control flow structuring algorithms that have taken inspiration from this early work (Section 4.4.2). Finally, some existing LLVM tooling is covered that might be useful to aid the control flow restructuring process for LLVM IR, especially concerning loops (Section 4.4.3).

4.4.1 Control Flow Graph Reducibility

Control flow analysis was pioneered in the 70s by Allen [1] who introduced the briefly mentioned reducibility property in Section 2.1.2.1. Reducibility is based on the notion of *interval analysis*. Allen defines an interval $I(h)$ in a CFG as "the maximal, single entry subgraph for which h is the entry node and in which all closed paths contain h". To illustrate this further, consider the graph in Figure 4.1. Some example interval of this graph include $I(0) = \{0, 1, 2, 3\}$, $I(2) = \{2, 3\}$, and $I(4) = \{4, 5, 6\}$.

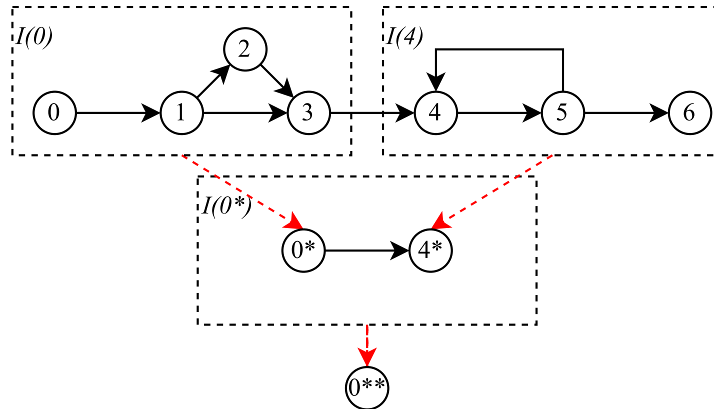


FIGURE 4.1: Example of a reducible control flow graph and its reduction into a single control flow node.

A graph G may be partitioned into a unique disjoint set of intervals g_i such that $\bigcup_i g_i = G$, and $\forall_{i \neq j} g_i \cap g_j = \emptyset$. By replacing the intervals with single nodes in the graph, a *first-order graph* can be created. This process can be repeated to create nth-order graphs until one of two things will happen:

1. The graph is reduced to a single node in which case the graph is said to be reducible.
2. An nth-order graph cannot be partitioned further to create a $n + 1$th-order graph in which case the graph is said to be irreducible.

VerCors requires reducible programs (or preferably, structured programs) for them to be verifiable using Hoare logic. LLVM IR however does allow for irreducible control structures, but fortunately has mechanisms to fix irreducibility that are discussed in Section 4.4.3. Alternatively, Cocke et al. suggest an algorithm that can transform any irreducible program into a reducible program by means of *node splitting* [14].

4.4.2 Existing Decompilers

Other than the previously mentioned C Backend (Section 4.3.1) that converts LLVM IR into C code, several other decompilation tools exist, such as Phoenix (Section 4.4.2.1) and DREAM (Section 4.4.2.2). They are not necessarily related to LLVM IR decompilation but do have interesting control flow structuring algorithms which can be relevant for VCLLVM.

Another recent decompiler called revng-c [25] features a structuring algorithm called control flow combing. However, it is questionable whether it has relevance to VCLLVM as it favours restoring idiomatic C code over the preservation of the original CFG. Significant modifications to the CFG might obstruct the origin traceability of verification errors which would make VCLLVM harder to use.

4.4.2.1 Phoenix

Phoenix [60] is a decompiler and converts x86 binaries to C code. To recover a structured program, it uses *structural analysis.*

Structural analysis [61] is an extended and improved version of the interval analysis technique mentioned in Section 4.4.1. It first attempts to recover more specialised control structure patterns and only uses intervals as a last resort. This is arguably more useful in the context of decompilation as instead of only being able to prove or disprove reducibility, structural analysis can be used to construct (or at least parts of) a structured program. An example of how such an algorithm might restructure a CFG can be found in Figure 4.2.

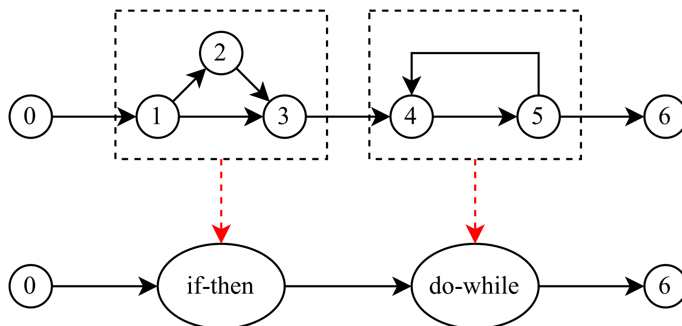


FIGURE 4.2: Example iteration of an arbitrary structural analysis algorithm on the same CFG as Figure 4.1.

Structural analysis can be implemented as an iterative algorithm where subgraphs of the CFG are replaced each iteration until a single sequence of nodes appears or the algorithm runs out of replacement options. In the example case of Figure 4.2, no nested control structures were present thus only one iteration was necessary but in theory, structural analysis can recover nested structures as well.

Phoenix made a novel contribution to structural analysis by preemptively introducing goto statements into the CFG. While this may seem counter-intuitive (the goal of structural analysis is to reduce the number of necessary goto statements after all), the introduction of a goto statement effectively replaces an edge in the CFG. This means that a recognisable structural pattern can appear and that the algorithm can continue resulting in fewer goto statements in the end result of the algorithm.

4.4.2.2 DREAM

DREAM [73] is a decompiler that takes a novel approach to structuring by moving away from structural analysis. DREAM recognises that one of the major shortcomings of structural analysis is that it is dependent on a fixed set of control structure patterns. This works under the assumption that the set of patterns is complete. However, as soon as the CFG presents a pattern unknown to the structuring algorithm, it results in unstructuredness in the form of goto statements.

Therefore, DREAM opts for an approach which it calls *pattern-independent structuring*. With this approach, it claims its output to be goto statement free. The algorithm starts by identifying cyclic and acyclic regions. Cyclic regions are similar to the connotation of loops in LLVM (Section 4.4.3) and are led by a single header into which backedges of nodes in the loop region lead. Acyclic regions have a single entry and a single exit node and no such backedges.

After regions have been identified, DREAM will start structuring the AST by traversing the CFG in post-order. Post-order traversal makes sure that a node is only processed after all the descendants are processed. This makes sure the AST is constructed with a bottom-up approach where nested control structures are resolved first.

The algorithm treats acyclic and cyclic regions distinctly.

For any acyclic region, DREAM calculates its *reaching condition* (i.e. a Boolean formula that conditionally describes the paths in the CFG leading to the region) for the cyclic region in which it is contained (or the root of the CFG if the acyclic region appears directly in the function body).

For cyclic regions, DREAM will first transform the region into a general form that makes sure that the loop region has a single entry and a single exit. Initially, the loop region is modelled as an infinite while-loop. Any edge to the exiting node of the loop is modelled as a `break` statement. Since a post-order traversal for structuring is used, it cannot be present inside a nested loop as the nested loop would already have been resolved in the AST. Therefore, the algorithm does not have to account for multilevel loop breaks. Finally, a set of inference rules is used to transform the loop into a more idiomatic form.

Lastly, an important detail of the algorithm is that it switches between two modes: The pattern-independent structuring mode described above and a semantics-preserving transformation mode. As the name suggests, without the latter mode, semantics by the structuring algorithm would not be preserved. It is responsible for the readability of the AST such as merging reaching conditions but also to make sure that reaching conditions that have side effects do not get copied unsafely around the AST.

4.4.3 LLVM Loop Analysis Tools

Control flow analysis is not only useful for decompilers but also for compiler optimisation passes. Therefore, the LLVM Project has useful loop analysis tools and passes as it recognised the complexity and utility of analysing loops in LLVM IR. First and foremost, the built-in loop analysis pass offers useful information about blocks and their role in a loop. It does so in the terminology outlined in the LLVM documentation [52] and is visualised in Figure 4.3. The terminology includes the following:

- A **header** is a block that serves as an entry point into a loop. A block can only be the header of one loop and every loop only has one header. This means that every loop is uniquely identifiable by its header, even if a loop contains inner loops.
- A **pre-header** is a block that precedes a header and has an edge going into the header in the CFG. A loop can have multiple pre-headers.
- A **latch** is a block that is part of the loop and has an edge in the CFG going back to the header, effectively creating a cycle. The edge going back to the header is known as a **backedge**. A loop can have multiple latches.
- An **exiting block** is a block inside the loop that has an edge to a block that is outside the loop. The edge is referred to as a **exiting edge** and the destination of this edge is called an **exit block**. A loop can have multiple exiting blocks, exiting edges, and exit blocks.

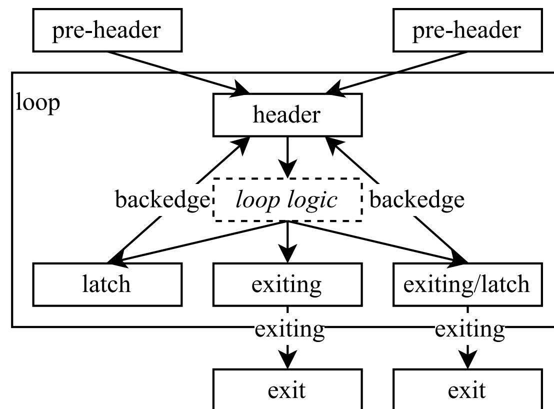


FIGURE 4.3: Example CFG of blocks featuring a loop where blocks are labelled with their corresponding loop terminology.

Despite the loop analysis passes aiding the uncovering of loops in the CFG, loop structuring itself is still by no means a trivial task. There are a few notes to be aware of:

- LLVM IR allows for irreducible control flow. This means that there are possible cycles in the CFG that are unresolvable to loops. For example, any cycle structure that has multiple entries (i.e. multiple headers) is irreducible. Fortunately, LLVM includes a `FixIrreducible` pass that converts irreducible control flow into loops. While this keeps the program semantically equivalent, the LLVM IR can drastically change syntactically. This in turn might hinder communicating the origin of errors to the user as the processed output does not resemble the original user input.

- Blocks in a loop can serve multiple functions at once as can be seen in Figure 4.3 where one of the blocks is both an exiting block and a latch. In theory, it is even possible for a loop to be a header, latch, and exiting block all at the same time. This actuality complicates loop structuring as there are many valid loop patterns. The variety can be partially mitigated by performing a loop rotation pass that has been built into LLVM. Loop rotation standardises all loops in the program to a do-while loop. The pass additionally adds a guard to the CFG before the loop when it cannot prove whether the loop body will be executed at least once.

  Loop rotations would inarguably reduce variations among loops thus aiding loop structuring, but it would share the problem of communicating the origin of errors to the user as is the case with the `FixIrreducible` pass.

Chapter 5

Tool Design and Architecture

This chapter sets out to provide an overview of the current design of VCLLVM as well as the considered alternatives. The chapter will first cover a rudimentary architecture outlining where and what critical design choices had to be made. This is followed by an enumeration of options for each design choice including an assessment of the advantages and disadvantages of each option. Finally, this section ends with a more concrete and complete design of what VCLLVM currently looks like.

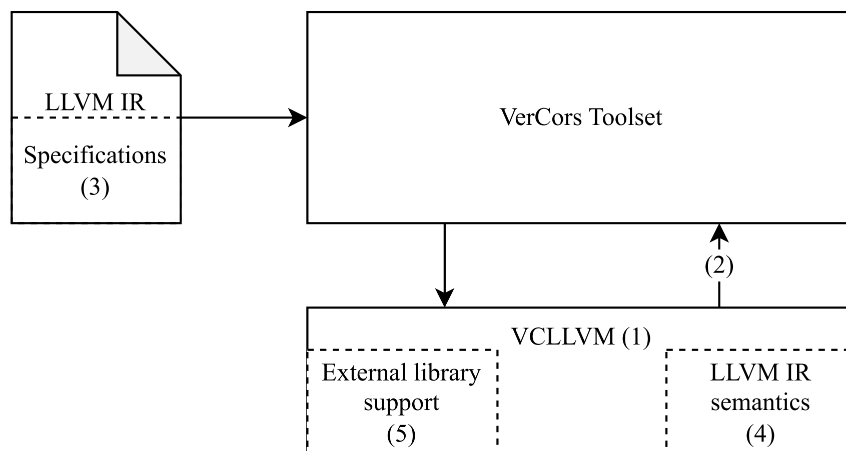


FIGURE 5.1: Rudimentary design of VCLLVM in relation to VerCors.

Figure 5.1 shows a rudimentary design of how VCLLVM fits into the VerCors Toolset. Each potential design choice has been labelled 1 through 5 and are the following:

1. **Embedding versus externalising**. The first choice to make is whether to embed VCLLVM into the VerCors codebase or develop it as an extension. Embedding could exacerbate the problems of Challenge 1 (3.2.1) while externalising would require serialisation to ensure interoperability between VerCors and VCLLVM. Serialisation comes with its own set of challenges. Sections 5.1 and 5.2 cover the design choices of embedding versus externalising and serialisation in more detail respectively.

2. **VCLLVM output**. When embedding the tool directly into VerCors, a COL AST is the obvious output format of VCLLVM. However, as mentioned in point 1, externalising VCLLVM would require serialisation. Since COL does not have a concrete syntax, there might be better options available. For example, PVL or Viper might be better candidates as these languages do have a concrete syntax. Design decisions on the output format of VCLLVM are covered in Section 5.2.

3. **Verification specification**. As discussed by Challenge 2 (3.2.2), parsing specifications and deciding on the syntax of the specifications are not straightforward procedures. Different syntax options for specifications are discussed in Section 5.3 while parsing implementation details are reserved for Section 6.2.

4. **LLVM IR semantics**. While most LLVM IR instructions are straightforward to capture semantically, there are a couple of exceptions that are much harder to deal with.

   For example, dealing with memory is challenging because memory management is very context and state-dependent. It gets even more complicated when concurrent memory operations come into play as outlined by Challenge 6 (3.2.6). The current iteration of VCLLVM does not yet cover any memory operations. Memory model design and assumptions are therefore reserved for future work and are discussed in Section 9.5.

5. **External library support**. LLVM IR is often compiled and linked against existing libraries to provide support for external libraries. An option would be to therefore not support any external libraries and consider their semantics unknown.

   However, as mentioned in Challenge 6 (3.2.6), to support any meaningful concurrency reasoning, there needs to be support for a concurrency library to deal with thread creation and signal handling. For interaction with heap memory, library support is also required. While this is part of the design, it has not yet been implemented and is considered future work. External library support is therefore discussed in Section 9.4.

5.1 Embedding vs Externalising

Embedding VCLLVM into VerCors or developing it as an external tool both have advantages and disadvantages. Both options will be separately discussed in this section.

5.1.1 Embedding VCLLVM

The main advantage of embedding VCLLVM into VerCors directly is the interoperability with existing VerCors code. There is no need for a communication protocol between VCLLVM and VerCors as the projects would co-exist in the same code base.

However, the main disadvantage is that VCLLVM would now be limited to the use of Scala as a programming language, or at the very least JVM-based languages. While this may not sound like a problem at first, it does mean that lots of tools available in the LLVM Project used for language analysis and parsing would not be available to VCLLVM as at the time of writing there is no (native) Java or Scala API for LLVM. This introduces a lot of extra work by having to develop all these tools from scratch.

Additionally, recall from Challenge 1 (3.2.1) that LLVM IR is not stable. This compounds the problem mentioned earlier in terms of maintainability. Keeping up with the rapid development of the LLVM Project would create a lot of technical debt and possibly introduce incorrect functionality as features get removed and added that might be overlooked during the maintenance of VCLLVM.

If VCLLVM were to be embedded into VerCors directly, there are a couple of existing tools that might help.

5.1.1.1 GraalVM and SuLong

GraalVM [71] is a *polyglot* virtual machine developed by Oracle Labs. GraalVM uses the Truffle API to accomplish its polyglot nature. With the Truffle API, a compiler can be developed that compiles to Truffle ASTs which in turn can natively run on GraalVM. Among the Truffle compilers is a Java bytecode compiler which means GraalVM can be used as a JVM.

Another Truffle compiler is SuLong [57] which offers an LLVM runtime environment on GraalVM. In other words, it compiles LLVM IR to a Truffle AST. This means that if VerCors were to run on GraalVM, it could leverage the Truffle API to obtain Truffle ASTs of LLVM IR code and transform that AST to an intermediate verification language such as COL or Viper.

However, apart from the obvious (and possibly unwanted) dependency on GraalVM as the only JVM for VerCors to run on, there are other caveats to look out for in this approach.

Firstly, not only would VerCors be dependent on GraalVM as a runtime environment, but VerCors would also become dependent on the development of SuLong. Which versions of LLVM VerCors would be able to support would be completely dependent on what version of LLVM SuLong supports. Given how at the time of writing the current release of GraalVM only supports up till LLVM 12.0.1[1] (a version that has been out since July 2021 [67]), the dependence on SuLong could imply to never be able to support the most recent versions of LLVM.

Secondly, it is important to note that by using SuLong, VerCors would technically support Truffle ASTs as a language and not LLVM IR. SuLong is essentially just another compilation backend for LLVM and would be comparable to supporting x86 compiled through LLVM instead of LLVM IR. This means foregoing the original goal of what VCLLVM set out to do as well as losing critical language features of LLVM IR. For example, it gives no guarantees that the compiled Truffle AST from a piece of LLVM IR code would have equivalent behaviour if that same LLVM IR code were compiled to another target.

For these reasons, a design including GraalVM and SuLong seems undesirable.

[1]See: `https://www.graalvm.org/22.3/reference-manual/llvm/Compatibility/`

5.1.1.2 Assembly Languages as Verification Target

Another approach to deal with the compatibility issues of LLVM IR is to bypass LLVM IR altogether in the verification process. This could be achieved by compiling LLVM IR to assembly code for a specific hardware target using an LLVM backend compiler. Preferably, it would be a compilation target that is maintained within the LLVM Project itself and has a relatively small instruction count.

For example, the RISC-V architecture would be a good candidate for such an approach. Formal verification of assembly languages or in this case, RISC-V, is not entirely novel [2, 3, 59]. It therefore at first glance seems like a possibility worth researching. Although this approach would have unrivalled compatibility and stability (after all, updating a hardware standard is comparably less trivial than updating a software standard), also this approach has some critical caveats.

Firstly, it has similar issues as the GraalVM/SuLong approach in the sense that also in this case, the VCLLVM would verify an LLVM IR compiler backend target rather than LLVM IR itself. It would suffer from the same issues as listed in Section 5.1.1.1.

Secondly, it would exacerbate the potential problems that Challenge 4 (3.2.4) already poses. Reintroducing levels of abstraction to LLVM IR is already posing a challenge but this would become even more difficult when using the RISC-V instruction set. For example, typing and function abstractions would be lost in the compilation process. Moreover, LLVM analysis tools such as pointer alias analysis, and CFG extraction would also not be available anymore.

5.1.1.3 JavaCPP

Using the native C++ API of LLVM has a lot of benefits in terms of development experience and available functionality. The drawback is having to interface C++ code with the Scala codebase of VerCors. JavaCPP might be able to provide such interfacing.

JavaCPP[2] is a project that aims to provide efficient access to native C++ code and libraries. Under the hood, it uses JNI[3] (Java Native Interface) which is a feature of Java to enable JVM code interact with native binary code of the platform. It defines and calls bindings to existing C++ (library) code. These bindings generate the required C++ code with JNI bindings to be compiled into a binary.

Because of its use of JNI, there is no dependency on a specific Java distribution as JNI is part of the Java Standard (unlike the GraalVM Truffle API (Section 5.1.1.1)). Additionally, JavaCPP provides a set of preset bindings for popular C++ libraries including LLVM[4] whose development seems to keep up with recent LLVM versions.

There is a problem however with this approach which is that JavaCPP becomes naturally unwieldy for big projects. This is caused largely by the fact that a significant part of the program state will reside inside the JNI binary interface instead of in the Java code itself.

[2]See: `https://github.com/bytedeco/javacpp`

[3]See: `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html`

[4]See: `https://github.com/bytedeco/javacpp-presets/tree/master/llvm`

Firstly, this makes the programs harder to debug as the state of an object is not exposed to the Java Runtime Environment and therefore requires explicit memory access by the debugger.

Secondly, the JVM garbage collector has no control over the memory state of the binary interface meaning that memory will have to be managed manually in Java. Java was not designed with manual memory management in mind which makes the endeavour even more error-prone than it already is in other languages such as C and C++.

Thirdly, JNI API calls come with overhead that can significantly hamper performance. The overhead is caused by having to communicate with the binary interface. This overhead is negligible in the case of long computational tasks. However, in the case of JavaCPP, every single function call would have to go through the JNI interface. By using lots of function calls for lots of smaller external functions, the JNI overhead could contribute to a significant portion of the total execution time of external functions and may cause serious performance issues.

Therefore, JavaCPP does not seem like a good candidate to build an entire project on. At best, JavaCPP could be used as a communication protocol between VerCors and VCLLVM while the source of VCLLVM itself will mostly be developed in C++. This option is further discovered in Section 5.2.2.1.

5.1.2 Externalising VCLLVM

Seeing how the proposed methods to embed VCLLVM inside the VerCors codebase (Section 5.1.1) all seem to have significant drawbacks, separating VCLLVM from VerCors to build an external LLVM tool seems enticing. Developing VCLLVM as an external LLVM tool has significant advantages over embedding it in VerCors itself:

- Parsing LLVM IR with LLVM itself guarantees consistency between LLVM IRs in-memory representation and textual representation. It thereby largely solves Challenge 1 (3.2.1).
- Analysing LLVM IR with LLVM itself guarantees consistency of analysis. Even if the analyser were to make a mistake in its analysis, backend compilers would be incorporating the same analysis mistakes in their compilation thus at least guaranteeing consistency.
- Developing VCLLVM using LLVM itself would save development costs. Many of the tools needed (such as analysis tools and a parser) would already be available and would not have to be redeveloped if another platform were to be used without native LLVM integration.

There are two significant drawbacks to developing VCLLVM as an external tool. The first one is that it will be harder for VCLLVM to communicate with VerCors. Possible options to deal with this issue are discussed in detail in Section 5.2.

The second drawback is the possible complications of specification parsing. Normally, VerCors supports any form of nesting between the specification language (first-order logic) and the source language as it can recurse back and forth between the specification language grammar and the source language grammar. This is an absolutely essential feature to support specifications that have nested first-order logic expressions mixed with source syntax. Possible options to deal with this issue are discussed in detail in Section 5.3.

5.1.2.1 Language Choice

Because the LLVM Project is entirely written in C++, C++ would be the obvious language choice for VCLLVM. The LLVM C++ API has also been a popular choice for other verification tools concerning LLVM such as LLMC (Section 4.2.1.1), FauST (Section 4.2.2.2), SMACK (Section 4.3.2), and SeaHorn (Section 4.3.4). However, the Haskell bindings of LLVM (llvm-hs) have also seen usage in verification tools. Notable examples include SAW (Section 4.2.2.3) and LLVMVF (Section 4.3.3).

However, other than potential personal preferences for the Haskell language, there seem to be only disadvantaged when choosing Haskell over C++. llvm-hs is a third-party implementation and is thus not part of the LLVM Project. This means the following:

- llvm-hs will always be slightly behind in updates as the maintainers will need some time to implement every new update to LLVM. This in turn will further slow down maintenance for VCLLVM if it were to be developed in Haskell.
- llvm-hs potentially does not cover the full LLVM API. llvm-hs themselves state that the set of bindings is "relatively complete"[5].

For these reasons, developing VCLLVM in C++ seems to be the best option.

5.2 VCLLVM Output Format

If VCLLVM were to be developed as an external tool, it would mean that the AST generated by VCLLVM would need to output a format that is either already interpretable by VerCors or a format for which an interpreter would be simple to implement in VerCors. Two output options will be discussed in this section. The first option is to use a concrete syntax (Section 5.2.1) and the second option is to serialise the COL AST (Section 5.2.2).

5.2.1 Using a Concrete Syntax

The most straightforward approach is to make VCLLVM output concrete syntax of an intermediate language. Preferably, it would be an intermediate language for which VerCors already has a parser available. As shown in Figure 5.2, LLVM IR would serve as input for VCLLVM and emit code in a language that could be parsed and processed by VerCors. Three languages will be discussed in this section which are C, PVL, and Viper.

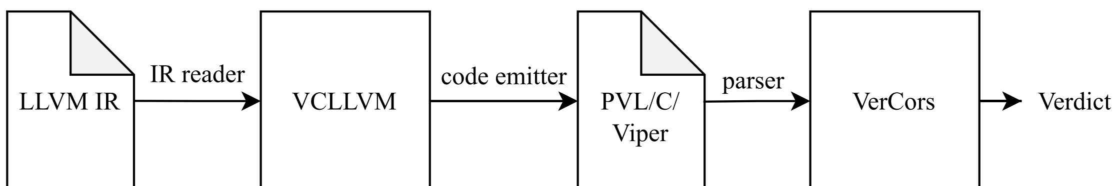


FIGURE 5.2: Input/output flow of VCLLVM and VerCors when using an intermediate language with a concrete syntax.

[5]See: `https://github.com/llvm-hs/llvm-hs/tree/5cf5911f98ee429bf646a88f5af63ef44705006c`

C seems like a sensible candidate for an intermediate language. After all, C is a language that is both supported as an output of LLVM through the C backend (Section 4.3.1) and as an input to VerCors. However, the C backend is poorly maintained. Moreover, the C verifier in VerCors does not support all language features. Both ends would require a lot of work to be able to make this approach feasible.

PVL seems like a more sensible candidate. While it would mean developing an LLVM backend compiler for PVL from scratch, PVL on the side of VerCors is actively being maintained. Also, VerCors has full control over the PVL language specification which means that there would be a lot of freedom to add features to the language to make the compilation from LLVM IR to PVL possible. These new features would also be essential as concepts like pointers are currently not supported by PVL.

Viper seems less viable compared to PVL. Not only would an LLVM backend compiler need to be developed from scratch, but Viper also has several disadvantages compared to PVL. For instance, Viper is less expressive than PVL, as it supports fewer types as well as fewer built-in functions and operators. This problem is further exacerbated by having less control over the language specification itself since it is not part of VerCors but the Viper verifier.

5.2.2 Serialising COL

PVL seems to be the best compilation target that has a concrete syntax. However, PVL is still just a subset of COL. Compiling to COL would still be preferable if it would somehow be possible to transfer the COL AST from VCLLVM to VerCors.

This is where serialisation comes in. Encoding the AST with VCLLVM and decoding it again with VerCors would enable the use of COL directly. Two options are considered in this section: Developing JNI binding for COL (Section 5.2.2.1) and encoding COL into Protocol Buffers (Section 5.2.2.2).

5.2.2.1 COL JNI Bindings

JNI allows JVM applications and native binary applications to interact with each other. JNI development requires development efforts on both the native application and the JVM application as they need to explicitly specify that their code is available and callable through JNI respectively.

This realisation has given rise to automation tools that only require coding on one side of the JNI bridge and automatically generate the code required to build said JNI bridge.

The aforementioned JavaCPP project is an example of such an automation tool which takes a JVM-first approach to the JNI bridge generation. Java code can be annotated to describe mappings to existing C++ code (such as classes, data structure types, functions, and more). Then on compile time, JavaCPP will automatically generate the required JNI code to interact with the precompiled C++ code.

On the other side, the Simplified Wrapper and Interface Generator (SWIG)[6] takes a native-first approach to the generation of the JNI Bridge. SWIG is a C/C++ tool that can be used twofold: The developer can either annotate header files with SWIG annotations or write special interface files that specify what parts of the program should be exported.

[6]See: `https://www.swig.org/`

SWIG can use this interface information to generate wrapper code for various programming languages including Java. It will generate the JNI code that if linked and compiled alongside the project it is wrapping, will allow the JVM to interact with the native code.

Both these tools could be used to encode and decode COL ASTs through a JNI bridge with a slight preference for the use of SWIG as SWIG delegates the interface development more to VCLLVM than to VerCors. Rather than having to retrofit the VCLLVM interface into VerCors, a VerCors interface could be designed on the clean slate that is VCLLVM.

A drawback of the JNI approach in general is that it still requires a lot of boilerplate code even when using tools like SWIG and JavaCPP. The reason is that for each COL AST node type that could be exported by VCLLVM, an interface would have to be declared in VCLLVM as well as an interpreter of the node type in VerCors. The boilerplate required may or may not be simple enough to automate as well.

5.2.2.2 COL Protobuffers

Protocol Buffers[7] (or Protobuf for short) offer a serialisation method that is largely automatable. Protobuf aims to provide language-neutral serialisation for data structures. Protobuf is interesting for VCLLVM for the following reasons:

- It has wide language support including support for Scala[8] and C++. This is essential for its integration in VerCors and VCLLVM respectively.
- It supports both code generation from and to a Protobuf definition. This will simplify the development of the communication layer between VerCors and VCLLVM considerably as both the generation of a COL Protobuf definition and the C++ implementation of COL could be fully automated.
- It provides versioning and update mechanisms that will not interfere with compatibility. This is a great feature to have since the language definition of COL itself is ever-changing. Having the possibility to update the Protobuf definition of COL and its C++ implementation without breaking changes is certainly an advantage.

There are alternative tools to Protobuf such as Apache Avro[9] and Ice[10]. However, all provide the functionality mentioned above and so the choice between them seems inconsequential.

5.3 Specification Syntax Design

As mentioned by Challenge 2 (3.2.2) designing a specification syntax comes with significant challenges for LLVM IR which are further exacerbated when externalising VCLLVM as mentioned in Section 5.1.2.

The first step is embedding the specifications into LLVM IR code such that they do not change the behaviour of the program but are available to VCLLVM after the LLVM IR program has been parsed. Since comments are ignored by the parser, the only option available is using LLVM metadata to embed specifications. An example of what that might look like can be found in Listing 5.1.

[7] See: https://developers.google.com/protocol-buffers
[8] See: https://scalapb.github.io/
[9] See https://avro.apache.org/
[10] See https://zeroc.com/products/ice

For readability, the specification contracts have been detached and written below the function, but in theory, they can also be inlined with the function definition. Listing 5.1 outlines three possible syntaxes for specifications. Each has its advantages and disadvantages which will be discussed here.

LISTING 5.1: Example embedding of specification contracts using LLVM metadata with three possible specification syntaxes.

```
1  define i32 @addMult(i32 %x, i32 %y, i32 %z)
2  !VC.contract !1 ;, !2 or !3
3  {
4    %1 = mul i32 %y, %x
5    %res2 = add i32 %1, %z
6    ret i32 %res2
7  }
8  ; (1) instruction block contract
9  !1 = !{
10   !"ensures",
11   !"%var1 = mul i32 %y, %x",
12   !"%var2 = add i32 %1, %z",
13   !"%verdict = icmp eq i32 %var2, %res2"
14 }
15 ; (2) independent specification language contract
16 !2 = !{
17   !"ensures %x * %y + %z == \result;"
18 }
19 ; (3) hybrid contract
20 !3 = !{
21   !"ensures icmp(eq, add(mul(%y, %x), %z), \result);"
22 }
```

The *instruction block* approach (from line 8) is to use a syntax that tries to stay as close to the LLVM IR syntax as possible. In the general case, it is a good idea to keep the syntax of expressions in the specification language as close to the syntax of the source language as it reduces possible ambiguity between source and specification. In the context of VCLLVM, this has both advantages and disadvantages:

- **Advantage**: As mentioned earlier, the ambiguity between the syntax of the source language and the specification language will be minimal.
- **Disadvantage**: The developer experience of writing and reading specification expressions will be suboptimal as the developer would need to write several instructions even for very simple expressions. Moreover, this makes writing specifications error-prone. However, one might argue the developer experience is not that relevant for VCLLVM as in the future it will most likely be used as an intermediate tool.
- **Observation**: Parsing the specification would have to be done in VCLLVM as specification expressions should be treated as small LLVM modules. Also, it would require the expression to be wrapped in a function as otherwise, it would reference registers that are out of scope.

The *independent specification language* approach (from line 15) would be to write specifications in a syntax that focuses entirely on readability and usability. The design of this language would be totally independent of LLVM giving a lot of freedom in the syntax. This approach also has both advantages and disadvantages:

- **Disadvantage**: It would create possible ambiguity between source and specification as both use a totally different syntax.
- **Advantage**: It would offer a good developer experience for writing and reading specifications as the syntax would flow perfectly with the existing specification language used by VerCors.
- **Observation**: Parsing the specification would require the least effort if it was done by VerCors itself. This is because VerCors already contains a lot of the building blocks needed to parse this kind of specification such as the lexer and parser grammar for VerCors base specification language. However, at the point where VCLLVM hands over control back to VerCors, only the program has been parsed into COL with the specifications still remaining. It is critical that the COL AST contains enough mapping information for all specification elements (such as function calls, variables etc.) in an unparsed specification to be still traceable.

The *hybrid* approach (from line 19) attempts to find a balance between the two previous approaches by trying to be both readable and unambiguous. Although it does neither job perfectly, it does support both to an acceptable level:

- **(Dis)advantage**: The syntax is relatively unambiguous as it uses the same keywords as the source language but does support nested expressions which is something LLVM IR cannot do.
- **(Dis)advantage**: It is relatively readable and writable as it follows a similar structure to the existing VerCors specification language. However, the syntax uses a lot of brackets which could cause trouble in keeping track of scoping.
- **Observation**: Like the *independent specification language* approach, this approach would benefit the most by being parsed by VerCors itself and it faces the same traceability challenges. Another complication is having to redefine large parts of the LLVM IR language into the specification language.

Although the *hybrid* approach does not excel at any particular point, it is also the only approach that does not offer any major readability or ambiguity issues. In combination with metadata variable aliasing and introducing unparsed specification COL nodes that include some form of mapping information to existing COL nodes in the extracted AST, this approach seems the most viable.

5.4 Design Summary

Having discussed the design choices and details in the previous sections, a reasonably complete and substantiated design can be created. The design as it stands can be seen in Figure 5.3.

In summary:

- VCLLVM is a separate tool from VerCors with its own code base written in C++. The added serialisation complexity does not weigh up against the usefulness of the analysis tools provided by the LLVM API.
- The input of VCLLVM is an LLVM IR file and the output is a Protocol Buffer that can be directly converted into a COL AST inside VerCors.
- The LLVM IR file provided to VCLLVM goes through several analysis passes to extract a semantically equivalent COL AST. The implementation of these passes is discussed in more detail in Section 6.1.
- The output Protocol Buffer by VCLLVM serves as input to VerCors. At this point, only the specifications still need to be parsed and resolved into the COL AST with the LLVM Specification Language Grammar (Appendix A). After the specifications are parsed, VerCors resumes as usual.

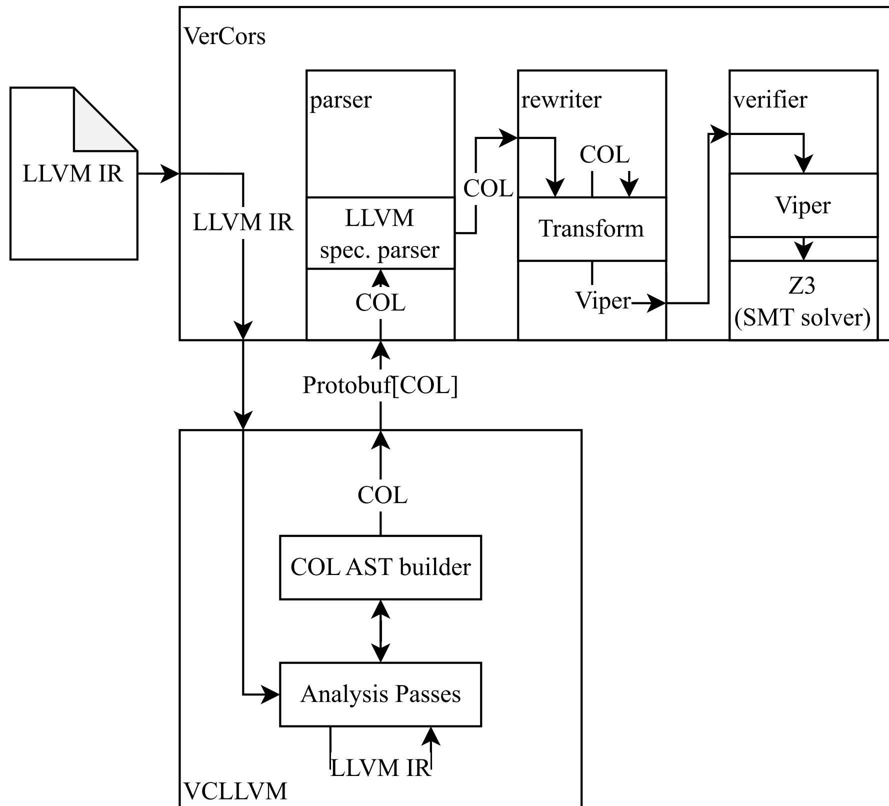


FIGURE 5.3: Full design of VCLLVM and how it fits in with the existing VerCors infrastructure.

Chapter 6

Tool Implementation

Chapter 5 gives a general overview of the design of VCLLVM. In contrast, this chapter is more focused on particular implementation details. Section 6.1 discusses the different analysis passes and their hierarchy. This is followed by a discussion of the two features that were most challenging to implement. These are covered in Section 6.2 and 6.3 and cover specification parsing and the origin system respectively. Finally, the regression test suite of VCLLVM itself is given some attention in Section 6.4.

6.1 Analysis Pass Hierarchy

VCLLVM uses several analysis passes to extract a COL AST out of an LLVM IR program. Their hierarchy is laid out in Figure 6.1. The direction of the arrows indicates the ordering in which the passes should happen (e.g. the *Function Declarer* pass should occur before the *Pure Assigner* pass).

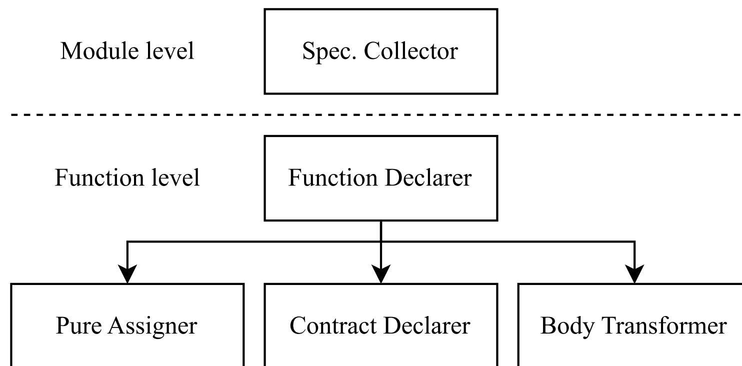


FIGURE 6.1: Pass hierarchy of VCLLVM with arrows describing the dependency relation.

There is a distinction between module-level passes and function-level passes. They indicate in which scope and what granularity a pass is run. In other words, a module pass is run on the module level and a function pass is run separately for each function in the scope of the function itself. The following passes were implemented:

- The **Spec. Collector** pass is responsible for collecting the specification metadata nodes on the module level. These are declared directly under the root of the AST as raw strings to be parsed into the AST by VerCors.
- The **Function Declarer** pass is responsible for declaring function definitions into the AST. It will also set the return type and arguments of the function.
- The **Pure Assigner** pass is a very small pass that checks whether a function is declared as pure and alters the function definition in the AST accordingly.
- The **Contract Declarer** pass is very similar to the Spec. Collector pass except it collects function contracts and adds them to the corresponding function definitions in the AST.
- The **Body Transformer** pass is by far the most extensive and complicated pass as it is responsible for setting the function body of a function definition. It is responsible for transforming the CFG of LLVM blocks into COL blocks but also transforming all the instructions within a block.

6.2 Specification Parsing

Apart from the challenge of choosing a suitable specification syntax as highlighted in Section 3.2.2 and 5.3, the implementation of how specifications are parsed is also interesting. To understand the problem VCLLVM faces when parsing specifications, consider the regular implementation of VerCors in Figure 6.2:

1. An input file is first parsed using the appropriate Antlr[1] grammar for the programming language in which the input file is written. This includes running a lexer and parsing the lexer tokens into an AST.
2. The resulting AST is then transformed into a COL AST (*ColLangParser* in the diagram). The result of this step is a COL AST that still contains language-specific nodes. This is intended to prevent losing language-specific semantics by simplifying nodes straight to the closest COL analogue.
3. The next step is to resolve unresolved references in the program. For example, the usages of declared variables and function calls are resolved in this step. This happens across the entire AST including cross-references between specifications and implementation.
4. Finally, the resolved COL AST is converted through rewriters into an increasingly stricter subset of COL nodes to prepare the program for verification. One pivotal class of rewriters are the language-specific rewriters that rewrite language-specific nodes to semantically (or as close as possible) equivalent COL analogues (*LangToCol* in the diagram).

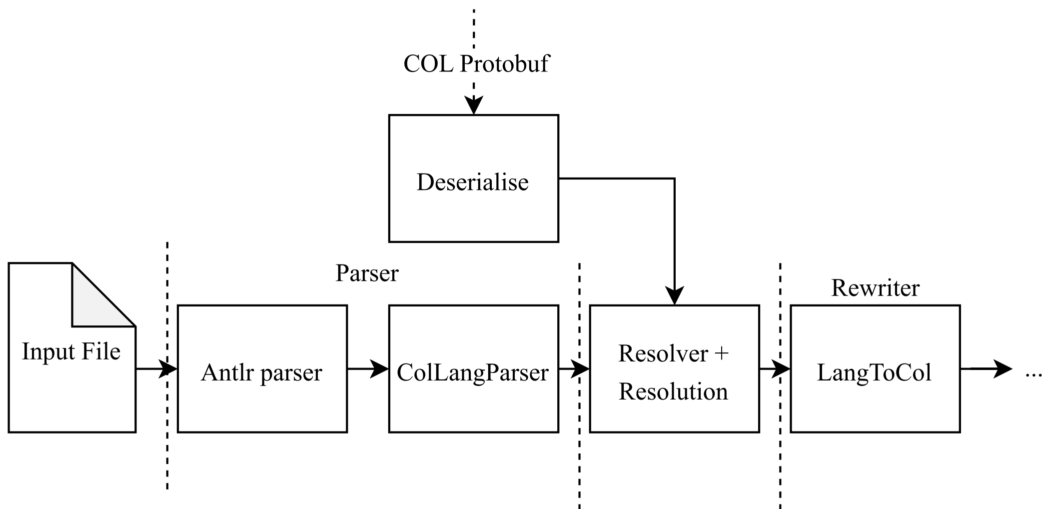


FIGURE 6.2: The usual implementation of parsing a program into VerCors, with the alternative *Deserialise* step included used by VCLLVM.

The crux of the specification parsing problem arises from the fact that all language-specific Antlr grammars inherit from one common specification grammar that dictates the grammar rules of all language-agnostic syntax. These include for example keywords like `requires`, `\result`, and the correct grammar rules of their usage.

With VCLLVM being an external tool, it becomes unclear whether it should be the responsibility of VCLLVM or VerCors to parse the specifications.

On the one hand, letting VCLLVM parse specifications is like reinventing the wheel as the entire Antlr parsing infrastructure present in VerCors would need to be reprogrammed into VCLLVM. This would not only include the specification parsers but also reprogramming any reference resolvers and seems like a lot of unnecessary work.

On the other hand, leaving the specifications unparsed and delegating the parsing to VerCors would inevitably delay parsing the specifications until the resolver and resolution step at the earliest. This is because the COL deserialiser (*Deserialise* in the diagram) serves as a replacement for the parsing chain.

The current implementation of VCLLVM opted to use the latter approach, in part because this approach has proven to be successful before in the implementation of JavaBIP verification [7]. For the JavaBIP implementation, JavaBIP annotations had to be translated into VerCors specification contracts first before they could be parsed into the COL AST facing a similar problem. JavaBIP annotations are therefore locally parsed as part of the reference-resolving process.

[1] Antlr (**AN**other **T**ool for **L**anguage **R**ecognition) is a powerful parser generator and therefore a popular compiler frontend. For more info on Antlr, see: `https://www.antlr.org/`

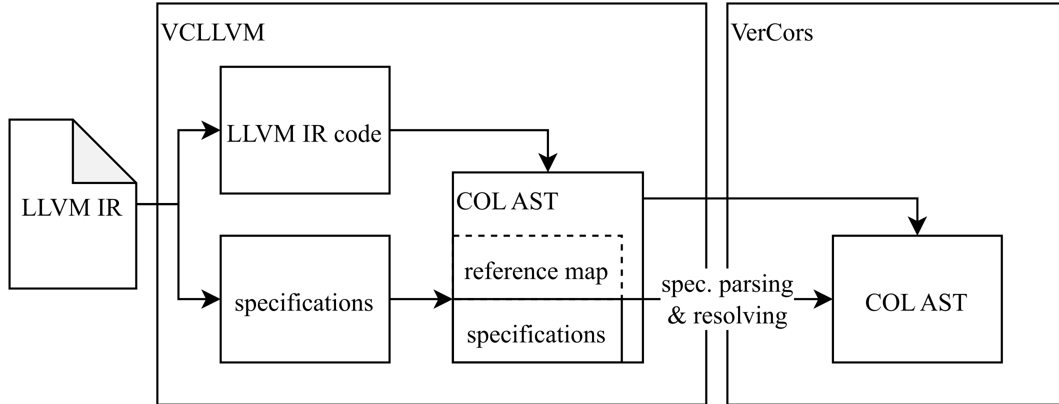


FIGURE 6.3: Specification parsing flow for VCLLVM.

This approach inspired a similar approach for LLVM specifications and is shown in Figure 6.3:

1. Specifications are encoded in LLVM IR metadata nodes and labelled accordingly. For example, there are `!VC.contract` and `!VC.globals` nodes for encoding function contracts and global specifications accessible to the entire module respectively.

2. The aforementioned metadata nodes are processed in a separate LLVM compiler pass and inserted in the correct locations in the COL AST.

3. The specification nodes that are produced in the COL AST can contain several reference maps. As an example, a `!VC.contract` metadata node is transformed into an `LlvmFunctionContract` node. The node contains a string of the raw, unparsed contract string and additionally two reference maps:

   (a) A variable reference map that maps the lexical representation of function arguments to a reference to the actual variable node representing the argument in the COL AST.

   (b) An invocation reference map that maps the lexical representation of functions (i.e. function names) to references to the actual function definition nodes representing the function in the COL AST.

4. Later, when VCLLVM has finished constructing the COL AST and the AST has been deserialised in VerCors, VerCors naturally resumes its reference resolving step. The resolver works by recursively processing the AST from top to bottom with a set of rules on what to do for each AST node type. Sticking with the contract example from earlier, the resolver contains a rule for an `LlvmFunctionContract` that instructs VerCors to locally parse the contract into the AST.

5. At this point, the contract specifications have been properly parsed into the AST. All that is left to do is to resolve the references within the contract. This is simply done by resuming the resolver recursion by resolving the recently created contract rule where the two aforementioned maps will be available in the context of the resolver. When recursing further into the contract and coming across a variable or function call, the resolver will simply check said maps and resolve the reference by a lexical comparison of the map keys against the lexical representation of the node.

6.3 The Origin System

As mentioned in Challenge 3 3.2.3, communicating the origin of errors to the user via line and column numbers (as is traditional in error reporting of most text-based systems) is infeasible due to the limitations of the LLVM IR parser that is built in LLVM. Therefore, a best-effort approach has been devised for VCLLVM to provide useful origin information of user errors.

For starters, it is paramount that the origin system of VCLLVM is fully compatible with the origin system used in VerCors. In VerCors, every AST node must be associated with an origin object. Many different implementations of the origin base class (or traits as they are known in the Scala programming language) exist in VerCors, but a minimal implementation of an origin class only has four required fields which will all be covered in the subsections of this section.

6.3.1 The `prefferedName` Origin Field

The `preferredName` field indicates the approximate name a node should be referred to as. For example, in the case of variables, functions, and files, it could be their literal name, or in the case of a constant, the lexical representation of that constant. The `preferredName` is mostly used for debugging purposes and is for example used in some cases to generate a textual representation of the AST.

The implementation of this field in VCLLVM is in most cases straightforward by setting their `preferredName` to what they are colloquially known as.

6.3.2 The (`inline`)`Context` Origin Field

The `inlineContext` and `context` fields are closely related. Both give the original source text of the node. For example, for a function definition, this would be the function signature and the function body.

In many cases the `inlineContext` and `context` are identical. However, in some cases, it is useful to not only include the source text of the node itself but also of the surrounding context. In this case, the `inlineContext` will refer to the literal source text representing the node whereas the `context` gives broader contextual information as well. For example, for a single statement in a function body, it would in many cases be useful to include the surrounding statements as well. In this case, the `inlineContext` would represent the original source text of the statement node while the `context` would also include some of the preceding and succeeding statements as well.

Both types of contexts are often used to present error messages to the user. For example, consider the following error messages taken from VerCors:

```
(...)
---------------------------------------
    1
              [------------
    2  ensures \result == 0;
               ------------]
    3  int foo() {
    4      bar();
---------------------------------------
[2/2] ... this expression may be false
(https://utwente.nl/vercors#postFailed:false)
=======================================
```

Here the `context` of the origin of the error is used to present all visible source text in the error message whereas the `inlineContext` is used to indicate the exact node in the AST that is causing the error by marking the corresponding source text of that node.

Here is where the first origin problems are encountered for VCLLVM. As established earlier, relating objects produced by the parser back to the original LLVM IR file cannot be trivially done. When regenerating LLVM IR code from LLVM objects there is information loss in terms of extraneous whitespace characters and comments.

Despite this, VCLLVM still uses regeneration of LLVM IR code to represent its contexts as there currently does not seem to be a viable alternative. It is a best-effort solution which in most cases should still be useful as the resemblance to the original source text is in many cases still relatable or in most cases completely equivalent.

6.3.3 The `shortPosition` Origin Field

The purpose of the `shortPosition` field is to communicate an exact position of an error. When it concerns source code, this usually takes the form of `<filename>:<line number>:<column number>` and in a generated error by VerCors could look as follows:

```
=======================================
 At path/to/example/file.pvl:2:1:
---------------------------------------
(...)
```

The same problem as for the context fields (Section 6.3.2) applies here as well: Deriving the exact position of the source code of an AST node is nontrivial due to possible discrepancies between regenerated LLVM IR code and the original LLVM IR file.

Because of this, VCLLVM opts for a best-effort solution using web navigation inspired *breadcrumbs* (Figure 6.4). Breadcrumbs are commonly used on websites that have a nested page structure to tell the user where they are on the website.

[2]See: `https://support.microsoft.com/en-us/office/upload-and-save-files-and-folders-to-onedrive-a5710114-6aeb-4bf5-a336-dffa7cc0b77a`

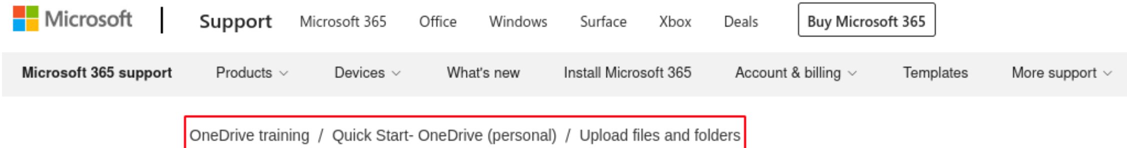


FIGURE 6.4: Example of a breadcrumb trail highlighted in red. Taken from the support website of Microsoft[2].

VCLLVM uses a similar approach to communicate code positions to the user and it uses the LLVM component hierarchy with the following format:

```
<filename> -> [function name] -> [block identifier] -> [instruction index]
```

And a `shortPosition` communicated to the user might look like this:

```
======================================
 At file path/to/example.ll
 -> function @foo
 -> block %bar
 -> instruction #1 (  %x = add i32 %y, %z):
--------------------------------------
```

While not as efficient as line and column numbers, this method should give the user enough information to locate a problem in the original source text.

6.4 Regression Testing

Regression testing in VCLLVM is implemented using *lit*[3] and its purpose threefold:

1. Only a small fraction of all LLVM instructions and features have been implemented yet. To guarantee support for new features will not break old features, it is good practice to employ regression testing.

2. When bumping up LLVM versions, there is a good baseline test suite to see if any breaking changes have been introduced that affect the functionality of VCLLVM.

3. VCLLVM should always stay compatible with VerCors and in particular its COL Protobuf interface. When updating the Protobuf definitions, regression testing can quickly determine whether VCLLVM is still compatible with VerCors.

How lit works and how it is implemented in VCLLVM is best explained by example. Consider the test file `test/C/arithmetics/02_int_add.c` in Listing 6.1.

[3]For more information on lit and the LLVM testing infrastructure, see: `https://releases.llvm.org/15.0.0/docs/TestingGuide.html`

Listing 6.1: **C** test program from the VCLLVM test suite.

```c
// RUN: clang -S -O2 -emit-llvm %s -o %t
// RUN: %VCLLVM %t
int int_add(int x, int y) {
    return x + y;
}

long long_add(long x, long y) {
    return x + y;
}

short short_add(short x, short y) {
    return x + y;
}
```

lit directives are specified at the top of a test file in comments of the programming language the test file is written in. lit supports different kinds of directives but the most used directive is the `RUN:` directive. The `RUN:` directive can be followed up by any shell command that will be run on the host machine. A test will pass if all `RUN:` directives are executed with a successful exit code.

The example test file contains two `RUN:` directives. The first `RUN:` directive instructs lit to compile this test C file (a file referring to itself is accomplished with the `%s` substitution) to LLVM IR using `clang`. It binds the output of `clang` to a temporary file with the `%t` substitution.

The second `RUN:` directive takes the output file of the last step (referring to the same `%t` substitution) and passes it on to VCLLVM. Note that like `%s` and `%t`, the VCLLVM binary is referenced with the `%VCLLVM` substitution. This substitution is custom configured for VCLLVM and will make sure that this substitution always points to the local and most recent build of VCLLVM.

Similar tests have been made and divided up into distinct categories to test certain features. Each category contains tests ranging from very simple examples to more complex examples in an effort to cover as many bases as possible. The following categories are currently present in the test suite:

1. The **types** category. It contains test files for each supported type. It only features very simple tests that test the type used as a variable, an argument, a return type, as well as constant initiations (e.g. `true` and `false` for Booleans). Every future supported type should have its own test file in this category.

2. The **arithmetics** category. It contains test files for each of the arithmetic operators (addition, subtraction, multiplication, and division) as well as a test with more complex nested arithmetic expressions.

3. The **branch** category. It contains tests for non-cyclic branching programming patterns. In the future, switch statement tests should also go into this category.

4. The **functions** category. It contains simple and complex tests concerning function calls. Complex cases include cyclic calls (i.e. two functions calling one another) and recursion.

As can be observed from the example earlier, VCLLVM currently only regression tests whether VCLLVM crashes or not, but does no further inspection or assertions on the output. The reason for this is twofold:

1. This method will catch most kinds of the errors enumerated earlier that VCLLVM at this level is concerned with. Whether they are changes in the LLVM IR specifications introducing new features that VCLLVM is unable to process or changes in the COL Protobuf definition, VCLLVM will likely error on any of these and therefore fail the test.

2. Capturing semantical assertions over the output of VCLLVM is nontrivial as there is no one correct formalisation of LLVM IR semantics into COL. Checking the output of VCLLVM semantically is a task that is much more sensible to delegate to the test suite of VerCors and is reserved as future work (see Section 9.3).

Chapter 7

Usage & Examples

This chapter functions as a demonstration of how VCLLVM is best used in its current iteration. The recommended usage of VCLLVM is covered in Section 7.1. Additionally, some verification examples of what VCLLVM is currently capable of are highlighted in Section 7.2.

7.1 Tool Usage

While writing LLVM IR code is possible and arguably slightly user-friendlier than writing raw assembly language, the fact of the matter remains that LLVM IR is an intermediate representation for compilers, not a programming language. Therefore, writing code to be verified by VCLLVM/VerCors in a higher level language than LLVM IR would be desirable.

It is therefore advised to write C code first, compile that C code to LLVM IR, and finally let VCLLVM/VerCors verify the LLVM IR program. C is recommended because the C LLVM compiler (`clang`) produces concise LLVM IR code (unlike some of the other frontends like `clang++` and `rustc`). Moreover, C is currently the only supported language in the regression test suite of VCLLVM. Of course, using any other language that has a compiler capable of outputting LLVM IR is possible as well, but the resulting LLVM IR program might contain LLVM IR syntax and library calls that are not supported yet by VCLLVM.

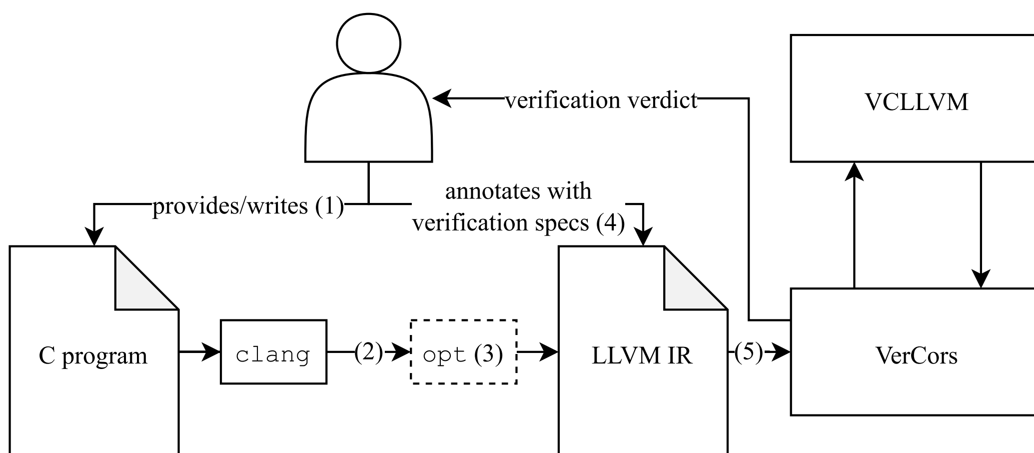


Figure 7.1: The recommended method for how to use VCLLVM/VerCors.

The recommended workflow to verify C programs is provided in Figure 7.1 and goes as follows:

1. The user first writes or provides a C program to be compiled with `clang`.
2. `clang`, with the correct compiler flags (`clang -S -emit-llvm (...)`), will output an LLVM IR file.
3. Optionally, the user can run the resulting LLVM IR file through the LLVM opt tool [53] to mitigate program structures VCLLVM cannot interpret. For example, the `-mem2reg` pass is occasionally in the test suite of VCLLVM to transform **`load`** and **`store`** instructions into register assignments wherever possible.
4. Once the user is content with the resulting LLVM IR file, the user can annotate the file with verification specifications whose syntax is explained in Section 5.3 and specified in Appendix A.
5. The annotated LLVM IR file can now be served directly into VerCors where the user will obtain a verification verdict on the program.

The user might want to iterate their specifications and implementation until a passing verdict is obtained. While changing the specifications is relatively straightforward, changing the implementation entails recompiling the program in which the specifications that were present in the LLVM IR file are consequently lost.

Ideally, a user would want to annotate their original C file and not be bothered by compiling and reinserting their specifications. However, making specifications persist through the compilation process poses significant technical challenges that are considered beyond the scope of this thesis. This persistence is therefore considered future work.

7.2 Verification Examples

This section presents three examples each highlighting different features of VCLLVM/VerCors. The first example (Section 7.2.1) focuses on the ability of VCLLVM/VerCors to reason about integer arithmetics. The second example (Section 7.2.2) focuses on branching and integer comparison operators. Finally, the third example (Section 7.2.3) focuses on function calls, in particular recursion.

Each example was obtained using the methodology described in Section 7.1. Moreover, some of the LLVM IR code obtained from compilation has been simplified and omitted for readability. All programs were compiled from C to LLVM IR with the following chain of commands:

```
$> clang -S -Xclang -disable-O0-optnone -emit-llvm <file>.c
$> opt-15 -S -mem2reg <file>.ll -o <file>.ll
```

where `<file>` represents the filename used without extension. Additional verification examples can be found in Appendix B.

7.2.1 Triangular Numbers and Cantor Pairs

The following example verifies the relation between *triangular numbers* and the *Cantor pairing* function. The two will first be briefly explained before diving into the actual example.

Triangular numbers are numbers that exist as a partial sum in the set of all natural numbers $\mathbb{N}$. The more formal definition adopted from Sloane et al. [62] is defined as:

$$T_n = \sum_{k=1}^{n} k = \frac{n(n+1)}{2} \qquad \text{where } n \geq 0 \tag{7.1}$$

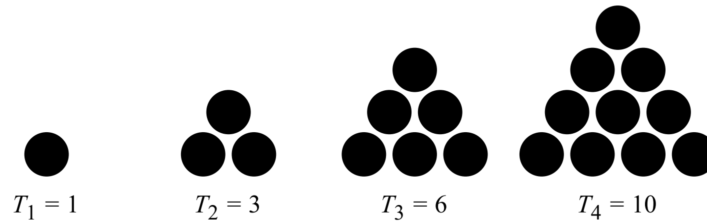


FIGURE 7.2: The first four triangle numbers (T_0 omitted) and their geometrical representation.

Their name is derived from the observation that when arranging circles into equilateral triangles, the total element count forms a triangle number. Some examples are shown in Figure 7.2.

Pairing functions are mathematical functions that can bijectively map $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$. The Cantor pairing function is a pairing function with the following definition:

$$cantor(x, y) = \frac{(x+y)^2 + x + 3y}{2} \qquad \text{where } (x, y) \in \mathbb{N}^2 \tag{7.2}$$

Proving that this function is bijective is beyond the scope of this thesis. For the interested reader, a proof of the bijection property is given by Lisi [40] and an "unpairing" function (i.e. a function that given a natural number can determine its original pair) is provided by Szudik [64].

However, plotting the pairing function in two-dimensional space can give some intuition on why the function is bijective. When observing Figure 7.3, it becomes apparent that both the set of integer coordinates in quadrant I of the two-dimensional space and the numbers produced by the Cantor pairing function (the set $\mathbb{N}$) have the same cardinality and thus a bijection between the two exists.

There is an interesting relation between the Cantor pairing function and the triangular numbers. When looking at the numbers on the line $y = 0$ in Figure 7.3, it seems the x coordinates directly correspond to the triangular numbers. This is trivial to prove by fixing y to 0 in the Cantor pairing function:

$$cantor(x, 0) = \frac{(x+0)^2 + x + 3 \cdot 0}{2} = \frac{x^2 + x}{2} = \frac{x(x+1)}{2} = T_x \qquad \square \tag{7.3}$$

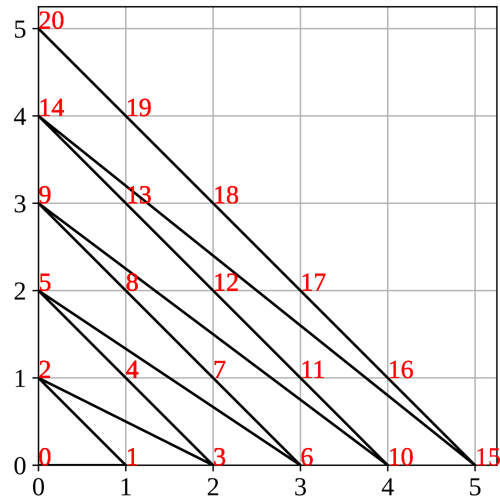


Figure 7.3: The first 20 Cantor coordinate pairs plotted and annotated with their corresponding Cantor pairing number.

This relation can also be proven using VCLLVM/VerCors as follows. For this, the following C program can be written:

Listing 7.1: **C** program with both the triangular number and Cantor pairing function.

```
1 int nthTriangular(int n) {
2     return (n * (n + 1)) / 2;
3 }
4
5 int cantorPair(int x, int y) {
6     return ((x * x) + x + (2 * x * y) + (3 * y) + (y * y)) / 2;
7 }
```

The functions are very similar to their mathematical counterparts except that in the `cantorPair` function, all squares have been expanded. Compiling the C program yields the following LLVM IR program:

Listing 7.2: **LLVM IR** program with both the triangular number and Cantor pairing function.

```
1  define dso_local i32 @nthTriangular(i32  %0)
2  !VC.pure !{i1 true}
3  {
4    %2 = add nsw i32 %0, 1
5    %3 = mul nsw i32 %0, %2
6    %4 = sdiv i32 %3, 2
7    ret i32 %4
8  }
9
10 define dso_local i32 @cantorPair(i32  %0, i32  %1)
11 !VC.pure !{i1 true}
12 !VC.contract !{
13 !"ensures icmp(eq, %1, 0) ==> icmp(eq, call @nthTriangular(%0), \result);"
14 }
15 {
16   %3 = mul nsw i32 %0, %0
17   %4 = add nsw i32 %3, %0
18   %5 = mul nsw i32 2, %0
19   %6 = mul nsw i32 %5, %1
20   %7 = add nsw i32 %4, %6
21   %8 = mul nsw i32 3, %1
22   %9 = add nsw i32 %7, %8
23   %10 = mul nsw i32 %1, %1
24   %11 = add nsw i32 %9, %10
25   %12 = sdiv i32 %11, 2
26   ret i32 %12
27 }
```

The specifications that have been inserted afterwards are marked in green. Both functions have been marked as *pure* (line 2 and 11) which means the following:

- The functions have no side effects (i.e. running the functions does not change the state of the heap).
- Consequently, because the functions do not have side effects, they can be called within verification specifications. This happens in line 13.

Furthermore, line 13 contains the property about the `@cantorPair` function that this example wants to prove. It states as a postcondition that when the argument `%1` is equal to 0, that implies (`==>`) that the value returned by `@cantorPair` is equal to the value returned by `@nthTriangular` evaluated on `%0`. This proof obligation poses a very similar proposition as the proof in Equation 7.3.

When running VCLLVM/VerCors on the example program in Listing 7.2, it is able to verify that this property universally holds.

7.2.2 Date Comparison

The following example is the verification of a simple date comparison algorithm. To demonstrate feedback on errors given by VCLLVM/VerCors, this example contains some intentional errors that VCLLVM/VerCors will discover and will be solved throughout the example. Consider the following C program:

LISTING 7.3: **C** program that checks whether a date occurs strictly after another date.

```c
1  #include "stdbool.h"
2
3  bool isLater(
4      int y1, int m1, int d1,
5      int y2, int m2, int d2)
6  {
7      if (y1 != y2) {
8          return y1 > y2;
9      } else if (m1 != m2) {
10         return m1 > m2;
11     } else {
12         return d1 > d2;
13     }
14 }
15
16 int test() {
17     isLater(
18         2023, 03, 07,
19         2020, 01, 01
20     );
21     isLater(
22         01, 01, 2023,
23         15, 03, 2023
24     );
25     return 0;
26 }
```

The program is quite simple and is primarily focused on demonstrating the ability of VCLLVM to deal with conditional logic. The program takes two dates as inputs where the first three arguments represent the first dates year, month, and day respectively. Similarly, the 2nd date is represented by the last three arguments.

As will be apparent later, VCLLVM can successfully transform if-then- (line 7), else-if-then- (line 9), and else-statement (line 11).

Compiling the C program yields the following LLVM IR program. Again, the added specifications have been marked green:

Listing 7.4: **LLVM IR** program that checks whether a date occurs strictly after another date.

```
1  define zeroext i1 @isLater(i32 %0, i32 %1, i32 %2,
2                             i32 %3, i32 %4, i32 %5)
3  !VC.contract !{
4  !"requires and(icmp(sge, %1, 1), icmp(sle, %1, 12));",
5  !"requires and(icmp(sge, %2, 1), icmp(sle, %2, 31));",
6
7  !"requires and(icmp(sge, %4, 1), icmp(sle, %4, 12));",
8  !"requires and(icmp(sge, %5, 1), icmp(sle, %5, 31));",
9
10 !"ensures icmp(sgt, %0, %3) ==> \result;",
11 !"ensures and(icmp(eq, %0, %3), icmp(eq, %1, %4))
12           ==> icmp(eq, \result, icmp(sgt, %2, %5));"
13 }
14 {
15   %7 = icmp ne i32 %0, %3
16   br i1 %7, label %8, label %10
17 8:
18   %9 = icmp sgt i32 %0, %3
19   br label %16
20 10:
21   %11 = icmp ne i32 %1, %4
22   br i1 %11, label %12, label %14
23 12:
24   %13 = icmp sgt i32 %1, %4
25   br label %16
26 14:
27   %15 = icmp sgt i32 %2, %5
28   br label %16
29 16:
30   %.0 = phi i1 [ %9, %8 ], [ %13, %12 ], [ %15, %14 ]
31   ret i1 %.0
32 }
33
34
35 define i32 @test() {
36   %1 = call zeroext i1 @isLater(i32 2023, i32 3, i32 7,
37                                 i32 2020, i32 1, i32 1)
38   %2 = call zeroext i1 @isLater(i32 1, i32 1, i32 2023,
39                                 i32 15, i32 3, i32 2023)
40   ret i32 0
41 }
```

When trying to verify the program the first error will become apparent:

```
[ERROR] Parsing file examples/private/datecompare.ll failed with exit code 1:
[VCLLVM] [Transform::Instruction::OtherOp] Return attribute "zeroext" not supported @
  file examples/private/datecompare.ll
 -> function @test
 -> block %0 (entryblock)
 -> instruction #1 (  %1 = call zeroext i1 @isLater(i32 2023, i32 3, i32 7,
  i32 2020, i32 1, i32 1))

[VCLLVM] [Transform::Instruction::OtherOp] Return attribute "zeroext" not supported @
  file examples/private/datecompare.ll
 -> function @test
 -> block %0 (entryblock)
 -> instruction #2 (  %2 = call zeroext i1 @isLater(i32 1, i32 1, i32 2023,
  i32 15, i32 3, i32 2023))

While processing "examples/private/datecompare.ll" VCLLVM has encountered 2 error(s).
Exiting with failure code...
```

In short, VCLLVM is complaining about the two function calls in the `@test` function using the **`zeroext`** attribute as it does not support this attribute. Since a word length of 1 bit (as is the case with the **`i1`** Boolean type in LLVM IR) is not supported by most CPU architectures, the **`zeroext`** indicates to the compiler that this type should be zero-extended to the next smallest word size supported by the architecture.

In some cases, this can make a semantic difference but has not been fully explored in VCLLVM. Therefore, any time the **`zeroext`** attribute is used in a function call, VCLLVM opts to throw an error. In this case, since the return value is not used further on in the function, zero extending the return value is inconsequential for the semantics of the program. Therefore, it can be safely removed from the program.

With the technical issues out of the way, VCLLVM/VerCors can now verify the specification of the program. The `@isLater` function does not do any sanity checks on the correct formatting of the dates itself. However, some preconditions for the input arguments of the function have been established:

- The month of the first date should be between 1 and 12 (line 4).
- The day of the first date should be between 1 and 31 (line 5)[1].
- The month of the second date should be between 1 and 12 (line 7).
- The day of the second date should be between 1 and 31 (line 8)[2].

[1]Technically, there are of course months with less than 31 days, but writing exact specifications for these is rather tedious and is not really the point of the example. There is a reason date-time libraries exist after all.

[2]Idem.

When attempting to verify the program, a precondition violation is detected:

```
======================================
 At file examples/private/datecompare.ll
 -> function @test
 -> block %0 (entryblock)
 -> instruction #2 (  %2 = call i1 @isLater(i32 1, i32 1, i32 2023,
  i32 15, i32 3, i32 2023)):
--------------------------------------
  %1 = call i1 @isLater(i32 2023, i32 3, i32 7, i32 2020, i32 1, i32 1)
      [-----------------------------------------------------------------]
  %2 = call i1 @isLater(i32 1, i32 1, i32 2023, i32 15, i32 3, i32 2023)
      [-----------------------------------------------------------------]
  ret i32 0
--------------------------------------
[1/2] Precondition may not hold, since ...
--------------------------------------
 At examples/private/datecompare.ll:2:10:
--------------------------------------
    1  requires and(icmp(sge, %1, 1), icmp(sle, %1, 12));
               [----------------------------------------
    2  requires and(icmp(sge, %2, 1), icmp(sle, %2, 31));
                ----------------------------------------]
    3  requires and(icmp(sge, %4, 1), icmp(sle, %4, 12));
    4  requires and(icmp(sge, %5, 1), icmp(sle, %5, 31));
--------------------------------------
[2/2] ... this expression may be false
======================================
```

A closer look at the verification error will reveal that the second call to `@isLater` in the `@test` function contains malformed dates. Instead of using the intended `yyyy, mm, dd` format, it swapped the format around by using `dd, mm, yyyy`. Consequently, verification fails as there exists no 2023rd day in any month. This bug can either be fixed by swapping around the arguments or removing the call entirely. For this example, the latter option was used by commenting out the violating function call.

Finally, there are two postconditions that are to be verified by VCLLVM/VerCors regarding the functional correctness of the implementation:

- If the year of the first date is strictly larger than the year of the second date, then the result of the function evaluates to true (line 10).
- If both the month and year of the two input dates are equal, then the result of the function will be equal to the evaluation of $d_1 > d_2$ where d_1 and d_2 represent the days of the first and second date respectively (line 11).

The postconditions form by no means a complete specification of the function (e.g. a postcondition that considers the case where the years are equal but the months are not is lacking), but VCLLVM/VerCors is able to verify this example. Of course, this is only the case when previously discussed modifications from the original example in Listing 7.4 are implemented.

7.2.3 Fibonacci Sequence

The following example demonstrates the ability of VCLLVM to deal with recursion. More specifically, the example will verify a C implementation of the Fibonacci sequence. Britannica [45] defines the Fibonacci sequence as follows:

$$fib(n) = \begin{cases} 1 & 1 \leq n \leq 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases} \tag{7.4}$$

The Fibonacci sequence has been studied extensively in literature and many interesting properties have been discovered. For example, the growth rate of the sequence tends to grow at the rate of the golden ratio. This is a ratio with many occurrences found in the geometric proportions of spirals in nature such as snail shells and hurricanes.

In the context of this thesis, the Fibonacci sequence is interesting because of its recursive definition which can be programmed as follows in C:

LISTING 7.5: **C** program that calculates the nth number in the Fibonacci sequence according to the definition in Equation 7.4.

```
1  int fibonacci(int n) {
2      if(n > 2) {
3          return fibonacci(n - 1) + fibonacci(n - 2);
4      }
5      return 1;
6  }
```

Compiling the C program yields the following LLVM IR program. Again, the added specifications have been marked green:

Listing 7.6: **LLVM IR** program that calculates the nth number in the Fibonacci sequence.

```
1  !VC.global = !{!0}
2  !0 = !{
3  !"pure i32 @fib(i32 %n) =
4    br(icmp(sgt, %n, 2),
5      add(call @fib(sub(%n, 1)), call @fib(sub(%n, 2))),
6      1);"
7  }
8  define dso_local i32 @fibonacci(i32 noundef %0)
9  !VC.contract !{
10 !"requires icmp(sge, %0, 1);",
11 !"ensures icmp(eq, \result, call @fib(%0));"
12 }
13 {
14   %2 = icmp sgt i32 %0, 2
15   br i1 %2, label %3, label %9
16 3:
17   %4 = sub nsw i32 %0, 1
18   %5 = call i32 @fibonacci(i32 noundef %4)
19   %6 = sub nsw i32 %0, 2
20   %7 = call i32 @fibonacci(i32 noundef %6)
21   %8 = add nsw i32 %5, %7
22   br label %10
23 9:
24   br label %10
25 10:
26   %.0 = phi i32 [ %8, %3 ], [ 1, %9 ]
27   ret i32 %.0
28 }
```

This example introduces an unseen feature of VCLLVM/VerCors which is *ghost code* at the global program level. Ghost code is code that purely lives in the specification space of the program and is therefore only able to read the program state rather than alter it. In other words, ghost code can only express pure program behaviour.

Declaring ghost code in VCLLVM is a little bit awkward because named metadata nodes at the global level in LLVM IR cannot contain metadata strings. Fortunately, unnamed metadata nodes can. By putting the specification inside an unnamed metadata node (`!0` in the example) and bundling the unnamed metadata node with the `!VC.global` named metadata node, VCLLVM is able to extract labelled metadata strings on the global level still.

In this example, a purely mathematical function of the Fibonacci sequence is defined. This is needed so that it can be used in the contract of the `@fibonacci` function. One might argue that it would be simpler to annotate the `@fibonacci` function as pure. While it is true that the function has no side effects and could theoretically be converted into a pure function, there are some technical limitations in the current VCLLVM/VerCors im-

plementation that prevent this from happening when LLVM IR functions reach a certain level of complexity. Verifying complex programs in the current VCLLVM/VerCors implementation heavily relies on pure functions. This is similar to the work of Paganoni and Furia [46] which relies heavily on predicates to verify Java bytecode.

The `@fibonacci` function has been annotated with one precondition and one postcondition. The precondition states that $n > 1$ where n is the provided argument (i.e. the value at the index in the Fibonacci sequence that should be calculated). This is required because the used definition of a sequence cannot have a -1st or 0th element.

The postcondition states that for any argument $n \in \mathbb{N}$, the `@fibonacci` function should return a value that is equal to nth value in the Fibonacci sequence. This should be a trivial consequence of the function.

VCLLVM/VerCors can verify both the pre- and postcondition of the `@fibonacci` function.

Chapter 8

Conclusion

This thesis presented VCLLVM: A Transformation Tool for LLVM IR programs to aid Deductive Verification. VCLLVM enables VerCors to formalise and verify LLVM IR. It also set out to answer a central research question (Section 3.1), to reiterate:

What are the necessary steps to make the VerCors Toolset support and verify LLVM IR programs?

In the context of LLVM and VerCors, seven challenges were identified which were:

1. Instability of LLVM IR (Section 3.2.1)
2. Parsing Verification Specification (Section 3.2.2)
3. Origin of User Errors (Section 3.2.3)
4. VerCors and Intermediate Compiler Representations (Section 3.2.4)
5. Control Flow Structuring (Section 3.2.5)
6. LLVM Concurrency Model (Section 3.2.6)
7. `undef` and `poison` Types (Section 3.2.7)

Challenges 1 through 3 have largely been solved. Challenges 4 and 5 have been tackled but have not fully been solved. Finally, Challenges 6 and 7 have unfortunately not yet been touched upon and are fully reserved for Future Work (Chapter 9). The remainder of this chapter covers Challenges 1 through 5 and briefly describes related goals that have been accomplished.

Challenge 1 This challenge can be considered solved as the design decision to externalise VCLLVM enabled it to be fully built using all the tooling available in the LLVM Project (Section 5.1.2). This means VCLLVM is not reliant on changes in the specification of LLVM IR but in the API specifications of the LLVM Project tools which are more stable and change less radically. This should enable VCLLVM to stay up to date with future versions of LLVM relatively effortlessly.

Additionally, the communication bridge between VCLLVM and VerCors implemented with Protobuf is automatically generated from the COL AST source code of VerCors. In turn, the Protobuf compiler is used to generate a C++ interface for VCLLVM. The automation of the communication process is in essence what makes externalising VCLLVM viable as both the COL AST source code and VCLLVM functionality can be extended without causing complications in the communication between VCLLVM and VerCors.

Challenge 2 This challenge can be considered solved both in respect of syntax (Section 5.3) as well as technical implementation (Section 6.2).

The syntax might not be the most aesthetically pleasing and is at times troublesome to write. This is likely caused by the extensive use of parentheses nesting in the syntax. However, it does succeed in striking a fair balance between the readability of the syntax and the unambiguousness between the source and specification language.

The technical implementation of processing verification specifications can also be considered solved as the current solution meets current requirements of what users should expect to be able to write in their specifications. Moreover, the current solution is robust and extensible enough to likely meet future requirements.

Challenge 3 The origin system currently present in VCLLVM and as presented in Section 6.3 is not ideal but is arguably optimal with the tooling available in the LLVM Project. The breadcrumbs approach is sufficiently convenient to find a location in the LLVM IR code albeit a little more cumbersome to use than line and column numbers.

Challenge 4 As it turns out, most LLVM IR features have an analogue in COL that is trivial to implement such as functions and most binary operators. Some instructions can get tricky such as `phi` instructions where the assignment to the register association to the `phi` instructions need to be backpropagated through the AST at the corresponding blocks referenced by the `phi` instruction. In the end, VCLLVM can transform the following LLVM IR features:

- **Basic Types** such as Booleans (`i1`) and arbitrary width integers (`iN` where $N \geq 2$). Their use has been seen in all examples in Section 7.2.
- **Arithmetic Binary Operators** on integers such as `add`, `sub`, `mul`, and `s`- and `udiv`. These operators have mostly been highlighted in the Cantor pairing example shown in Section 7.2.1, particularly the deductive reasoning capabilities of VCLLVM/VerCors.
- **Comparison Operator** for integers (`icmp`) and all its possible predicates. These were mostly highlighted in the date comparison example of Section 7.2.2.
- **Control Flow Features** such as `br`, `label`, `phi`, `call`, and `ret`. This includes **Function Definitions** through the `define` keyword. Control flow was mostly highlighted in the date comparison example in Section 7.2.2 through various branching structures. The Fibonacci example in Section 7.2.3 on the other hand highlighted VCLLVM/VerCors flexibility in function use including recursion.

The currently supported feature set of LLVM IR is only a small subset of the vast set of features LLVM IR offers. Not all features are used as commonly, but two essential missing features are any kind of memory operations (e.g. `load` and `store`) and aggregate data types (e.g. arrays, structs, and vectors). An extensive feature road map is presented in Future Work (Chapter 9).

It is also worth noting that the current integer support is semantically incomplete. Current problems with the integer transformation and potential fixes are reserved as future work and covered in Section 9.2.1.1.

Challenge 5 The current control flow structuring algorithm implemented in VCLLVM only supports acyclic branching. Loops have been reserved as future work as they are tricky to implement which has been touched upon in Sections 3.2.4 and 3.2.5.

The current implementation of acyclic branch structuring could also be considered suboptimal as it still heavily relies on goto statements. This is because it takes the naive strategy of copying the CFG structure one-to-one into COL. As mentioned before, goto statements should be used cautiously as they can introduce irreducibility in the CFG of a program. However, since the current programs produced by VCLLVM are guaranteed to be acyclic, they are by extension guaranteed to be reducible. Therefore, the current goto statement usage is permissible.

Chapter 9

Future Work

This chapter sets out to lay out the future of VCLLVM and ideas on how to improve the current implementation. Section 9.1 will cover a road map of features that are still to be implemented in VCLLVM. The remainder of the chapter will go into more detail about some of the proposed features in the road map in addition to some suggestions to improve the test infrastructure (Section 9.3).

9.1 VCLLVM Feature Road Map

While VCLLVM can aid in the verification of simple programs at the time of writing, its full potential has not been reached yet. Table 9.2 presents a road map of features that could still significantly improve VCLLVM. The proposed features have been prioritised using the *MoSCoW* methodology [13] which is an acronym for **M**ust, **S**hould, **Co**uld, and **W**ill not. It gives four levels of priority of which the last is for features that are undesirable.

Additionally, some features have been annotated with a section number. In these sections of this thesis, certain implementation ideas, related research, or implementation pitfalls can be found on that feature.

Feature	**Subdivision**	**Relevant Syntax**	**Prio.**	**Sec.**
Types	Improved integers	iN where $N \geq 2$	1	9.2.1.1
	Voids	**void**	1	
	Floating points	**half**, **float**, **double**	1	
		fp128, **x86_fp80**, **ppc_fp128**	2	
Aggregates	Pointers	ptr	1	9.2.1.2, 9.5.1
	Arrays	[n x <type>]	1	
	Struct	{<type> , ...}	2	
	Vectors	<[vscale] x n x <type>>	3	
Casting	Integer <>integer	**trunc** .. **to**, **zext** .. **to**, **sext** .. **to**	2	9.2.1.1
	Float <>float	**fptrunc** .. **to**, **fpext** .. **to**	2	
	Float <>integer	**fptoui** .. **to**, **fptosi** .. **to**, **uitofp** .. **to**, **sitofp** .. **to**	2	

Feature	Subdivision	Relevant Syntax	Prio.	Sec.
Casting *cont.*	Pointer <>integer	`ptrtoint .. to`, `inttoptr .. to`	3	
	Agnostic cast	`bitcast .. to`	3	9.2.1
	Address space cast	`addrspacecast .. to`	3	9.2.1.2
Operators	FP binary operators	`fadd`, `fsub`, `fmul`, `fdiv`, `frem`	2	
	Bitwise binary operators	`shl`, `lshr`, `ashr`, `and`, `or`, `xor`	3	9.2.1.1
	Aggregate operators	`getelementptr`	1	
		`extractvalue`, `insertvalue`	2	
		`extractelement`, `insertelement`, `shufflevector`	2	
Branching	Loops	`br`, `switch`	1	4.4, 9.2.2
	Jump tables		2	
Functions	Control	`invoke`, `unwind`	3	
Memory	Heap	`load`, `store`, `@malloc`, `@free`	1	9.4, 9.5
		`@calloc`, `@realloc`	2	
	Atomic	`fence`, `cmpxchg`, `atomicrmw`	2	
Concurrency	Ordering	`seq_cst`	2	
		`unordered`, `monotonic`, `acquire`, `release`, `acq_rel`	3	
	Threads	`@pthread_create`, `@pthread_join`	2	9.4
	Locking	`@pthread_mutex_lock`, `@pthread_mutex_unlock`	2	
Specifications	Loop invariants	`!VC.loop_invariant`	1	
	Assertions	`!VC.assert`	1	
Exceptions	Types	`undef`	-1	9.2.1.3
		`poison`	3	
	C exception handling	`@llvm.eh.sjlj.setjmp`, `@llvm.eh.sjlj.long`, `@llvm.eh.sjlj.lsda`, `@llvm.eh.sjlj.callsite`	3	
	C++ exception handling	`invoke`, `unwind`, `landingpad`, `resume`, `@llvm.eh.begincatch`, `@llvm.eh.typeid.for`, `@llvm.eh.endcatch`, `@__cxa_allocate_exception`, `@__cxa_throw`	3	

TABLE 9.2: Feature road map of VCLLVM with priorities 1 = Must, 2 = Should, 3 = Could, and -1 = Will not.

9.2 Abstracting Low-level and LLVM-specific Constructs

While abstractions of low-level features of LLVM have been implemented reasonably successfully so far, there is still room for improvement in the current implementation as well as some problems to solve for future unimplemented features. Most notably, the current type system (Section 9.2.1) can still use a lot of improvements. Moreover, some changes in the control flow structuring algorithm are desirable (Section 9.2.2).

9.2.1 Type Semantics

9.2.1.1 Integers

VerCors currently only supports one integer type which is most accurately described as the set of all mathematical integers $\mathbb{Z}$ and is therefore unbound. Using this representation is sound in most cases when the assumption is made that overflows do not occur. While in languages such as Java and PVL used in VerCors, this assumption is somewhat reasonable, it causes the following problems for LLVM IR:

- While LLVM IR makes a distinction between signed and unsigned integers (such as having distinct **`sdiv`** and **`udiv`** operations for division), it does not type check on signedness for them (both an `int64_t` and `uint64_t` in C would both end up as an **`i64`** in LLVM IR). It is therefore allowed in LLVM IR to use signed operations on unsigned integers and vice versa. This is of course problematic as the result of these operations can be unpredictable.

- LLVM IR supports arbitrary width integers thus including very narrow integer ranges such as **`i4`** and **`i8`**. However rare they may be, on specific platforms they might still occur and can cause real problems if integer bounds are not checked.

- Bitwise operations will be cumbersome to implement. While shifting bits around could still be hacked together using division and multiplication, implementing any binary bitwise operators such as **`and`**, **`or`**, and **`xor`** become convoluted when implemented on mathematical integers. Moreover, any truncation or extension of integer width operators for type casting purposes have no analogue in the mathematical domain[1] and would therefore have to be forcefully ignored.

Fortunately, these problems are not insurmountable. However, it should be noted that there exists no silver bullet to solve these issues. Nonetheless, here are some suggestions:

- One approach would be to entirely encode integers as bit vectors into VerCors. This would result in the most accurate semantical representation and would allow for very precise semantical definitions of all integer operations. It would additionally solve all three aforementioned problems: Integer widths are directly represented in the length of the vector, signedness would just be a bit in the vector, and all bitwise and casting operations would be trivial to implement on a bit vector.

  However, it should be noted that proofs could become extremely expensive to generate for a solver as an integer that used to be just one variable got expanded into an entire vector of variables that all have to be covered individually in the proof.

[1] Floating points face a similar problem with width conversion as traditionally VerCors treats floating points as elements of the set of real numbers $\mathbb{R}$.

- A simpler approach would be to keep the LLVM integer type somewhere in between a bit vector and a mathematical integer in terms of abstraction. For starters, the LLVM integer type could be represented in VerCors as a compound type of a Boolean type to represent the signedness and a regular mathematical integer type to represent the absolute value.

  By adding a proof obligation for signedness on each integer usage in an LLVM IR program, proofs could be generated that could prove useful properties about signed operator usage. For example, VerCors could ensure no signed integer is ever used on an unsigned division and vice versa.

  However, also this solution has problems. Firstly, it does not solve any issues regarding representing integer width and applying bitwise and casting operations on integers. Secondly, the introduction of extra proof obligations could make generating proofs cumbersome. While in many cases, VCLLVM would be able to derive signedness from context, in other cases it can be hard to derive the signedness of an integer, consequently resulting in littering the code with assertions about signedness.

  Take for example constant integers. A constant integer of -1 does not necessarily mean that constant is supposed to be signed as it could also just be that the user is being clever in creating the max unsigned integer value. A similar argument could be made for any constant positive integer not necessarily being unsigned.

9.2.1.2 Pointers

Pointers are not yet supported by VCLLVM and do have an implementation caveat regarding typing. As briefly touched upon in Section 2.2.1.3, the LLVM Project is currently transitioning from typed pointers (i.e. pointers to memory that contain a type annotation) to opaque pointers (i.e. pointers to memory without any indication of the data type stored in memory). This allows for memory behaviour that might be difficult to semantically capture.

Listing 9.1: Valid **LLVM IR** code with unexpected type conversion through opaque pointers.

```
1 define i32 @IntPlusFloat(i32 %0, double %1) {
2   %d_ptr = alloca double
3   store double %1, ptr %d_ptr  ; store as double
4   %lt = load i32, ptr %d_ptr   ; load as i32 without type conversion
5   %var = add i32 %0, %lt       ; unexpected behaviour
6   ret i32 %var
7 }
```

Consider the example in Listing 9.1. Here, a `double` is stored in memory returning an opaque pointer. Seeing how the pointer has no associated type, LLVM IR takes no problem in loading the contents of the same memory address as an `i32`.

Semantically, this is equivalent to performing a `bitcast` instruction. As established in Section 9.2.1.1, a `bitcast` instruction could only be formalised correctly by the use of bit vectors.

Alternatively, the behaviour of performing bitcasting through opaque pointers could be disallowed completely by checking if memory values do not change types in-between loads and stores. This could be achieved by adding proof obligations for each instantiation of an opaque pointer regarding its type.

9.2.1.3 `undef` and `poison` Types

Challenge 7 (Section 3.2.7) discusses the potential problematic `undef` and `poison` type.

The `undef` keyword indicates an undefined value as a result of undefined behaviour. It is very problematic to define semantics for the `undef` keyword, because `undef` is indicative of having undefined but correct behaviour (as confusing as that may sound). In order, for VCLLVM/VerCors to be able to derive anything useful from `undef` values it would either...:

1. ...have to enumerate through all possible values of the `undef` value and prove all of them individually.

2. ...have to derive from context what the compiler is going to end up optimising the `undef` value to through proof obligations about the compiler, the operating system, the target hardware, and all other factors that may influence the final evaluation of a `undef` value.

Both options seem completely infeasible, either due to state space explosion in the former suggestion or the impossibility of deriving what each compiler option will transform the value to in the latter suggestion. In some niche cases, evaluating `undef` might still be possible (e.g. an `undef` Boolean only has at most two possible values). Therefore, it is argued that any `undef` values in a program should always result in a verification error as the default behaviour. Some niche, doable edge cases could still be handled (such as the Boolean example) with alternative verification behaviour. This will add incompleteness to VCLLVM but will at least make sure VCLLVM stays sound.

On the other hand, `poison` values are simpler to reason about as all `poison` value instantiations are semantically equivalent. Rather than `undef` representing a set of possible values, `poison` simply communicates an error state and all poison values are equivalent. This is comparable to how for example all `null` and `NaN` values in some languages are often equivalent.

Seeing how VerCors is already able to reason about `null` values even in proof specifications (For example, in Listing 2.9 in the form of null checks), similar strides could be made for `poison` values. It would arguably be very useful to be able to provide that the result of a method cannot be poisonous as a specification.

9.2.2 Control Flow Structuring

The current control flow structuring in VCLLVM is mediocre at best. It currently employs a naive strategy of mapping every block in an LLVM IR function onto a COL block with a label. This enables a faithful representation of the original control flow of the function by replacing every unconditional and conditional branch instruction with one or two goto statements respectively.

While this approach seems to suffice to verify acyclic functions, this approach is completely inept to deal with loops as it will not generate loop structures in the COL AST. It is crucial for cyclic structures to be converted into loops as without loops, there is no space for loop invariants which are in turn crucial for the verification of programs with loops.

Fortunately, restoring control flow structure is not a novel problem, as seen in Section 4.4. Structuring algorithms as presented by Phoenix (Section 4.4.2.1) and DREAM (Section 4.4.2.2) can offer a stepping stone for structuring passes inside either VCLLVM or VerCors. Whether to perform the structuring inside VCLLVM or VerCors is a tradeoff between having LLVM analysis tools available and ease of access to the COL AST respectively.

Another tool worth mentioning in this discussion is ByteBack [46]. ByteBack resembles VCLLVM a lot except it targets Java bytecode (instead of LLVM IR) and uses Boogie [5] as a verification backend (instead of Viper [43]). What makes ByteBack interesting is that it does not restore any control flow structure. However, it does support loop invariants despite naively mapping goto statements in the bytecode directly onto Boogie goto statements. It is therefore definitely worth investigating how Boogie supports loop invariants in unstructured programs and whether Viper (the verification backend used in VerCors) could achieve the same.

9.3 Extending Test Infrastructure

As can be read in Section 6.4, testing infrastructure in the form of regression testing is already present in VCLLVM. However, it only tests whether VCLLVM throws errors or crashes at any point during its runtime. It would unarguably also be useful to be able to test the correctness of its implementation. In other words, it would be useful to test whether the programs produced by VCLLVM are semantically consistent with the original program.

Since VCLLVM depends on VerCors to function anyway and VerCors already has its own test infrastructure in place to test correctness based on a collection of verification examples, it would make sense to integrate this into the VerCors test infrastructure. The tests work by having example programs with verification specifications and an expected verification verdict (i.e. either a pass or a fail with the type of failure).

While giving no guarantees about correctness, it does at least perform regression testing on correctness in that changes will not suddenly resolve into verification failures that should pass or vice versa. Given that the test examples are diverse enough, it should be able to discover most breaking changes.

Adding LLVM IR tests to the VerCors test suite seems appealing, but there are a few caveats that need to be resolved:

- Adding LLVM IR tests to the VerCors test suite would mean VCLLVM would have to become a hard dependency of VerCors. VCLLVM is simply not mature enough for this. For example, portability is an issue right now as it has currently only been tested to run on Linux with either `gcc` or `clang` as a compiler, and ninja or make as a build system through CMake.
- Preferably, the example programs would be written in C and compiled to LLVM IR as part of the test process, similar to how lit is currently used in VCLLVM to accomplish the same effect. This requirement comes with its own set of sub-problems:
  - Dependency-wise, it would require both `clang` and lit to be available to VerCors adding another layer of dependencies on top of VCLLVM itself. `clang` is already an existing VerCors dependency but lit is not. Alternatively, it might be possible to eliminate lit by programming similar functionality within the Scala test runner.
  - Since the VerCors test suite depends on verification specifications and these specifications are only inserted *after* the compilation step from C to LLVM IR, the specification would have to be injected as part of the test process. This is a nontrivial problem.

In conclusion, the current maturity of VCLLVM is holding it back from being integrated into the test suite of VerCors. In the end, once sufficient maturity is achieved for VCLLVM, it is up to the maintainers of VerCors to facilitate LLVM testing inside VerCors.

9.4 External Library Support

As programmers do not want to reinvent the wheel every time they start a new project, support for external libraries that accomplish common tasks are often supported by compilers. LLVM is no exception and even heavily relies on them as in some cases LLVM IR is not expressive enough to support certain behaviours.

Two notable and relevant missing features of LLVM IR are heap memory allocation, and thread creation and signal handling. Especially the latter is especially important in the context of VerCors as the VerCors toolset is specialised in the verification of concurrent programs. These two features could be handled by the C standard library and the POSIX pthread library respectively. Thus, both should be (partially) supported by VCLLVM to fill the missing gaps of LLVM IR in the future.

9.5 Memory Model

LLVM IR operates on memory on a low level and is very permissive on the operations that are allowed on memory. To make sure the memory model used is sound, it may mean that VCLLVM is more restrictive on its memory operations than LLVM to guarantee that all the behaviour it does support is at least sound. While any form of memory management has not yet been implemented in VCLLVM, this section discusses some implementation ideas that could be useful in the future.

9.5.1 Memory Layout

VerCors itself maintains an implicit memory model meaning heap variables do not have an address or layout to adhere to. Memory locations exist in some arbitrary part of memory and can expand indefinitely. This makes the memory model inside VerCors a lot simpler than the memory model of LLVM or on a hardware level for that matter.

While the VerCors model is a lot simpler to handle and manage, it does disallow some pointer behaviour that would be valid in LLVM but is impossible in VerCors. Most notably, when a pointer goes out of the bounds of an array or struct, it can still end up in a valid memory location and thus produce valid behaviour. However, in VerCors, there is no notion of a layout of memory, so it is impossible to know whether a pointer going out of bounds ends up in a valid memory location.

This means VerCors will have to generate an error when this kind of pointer behaviour is employed. One could argue not supporting this pointer behaviour is not a problem since letting pointers go out of bounds of the data structure they are assigned to is bad coding practice anyway.

Another issue concerning pointers, especially when the program is written in SSA, is that it is sometimes hard to determine whether two pointers point to the same memory location. For the VerCors memory model to stay sound with regards to the LLVM memory model, it is essential that two separate memory locations cannot get created in VerCors for two pointers while in actuality, they are pointing to the same memory location. Fortunately, LLVM has pointer analysis tools[2] which can aid the derivation of semantically correct pointer behaviour in VerCors.

9.5.2 Permissions

When interacting with memory, the assumption must always be that the program is running concurrently. This is especially true since LLVM has no notion of signal handling and thread creation and thus no awareness of concurrent executions. Every memory instruction must therefore have proper semantics regarding permissions. These are outlined in Table 9.3.

Operator	Permission semantics
`alloca`	Grants write permission
`load`	Requires read permission
`store`	Requires write permission
`fence`	Redistribute permissions
`cmpxchg`	Requires write permission
`atomicrmw`	Requires write permission
`getelementptr`	None

Table 9.3: LLVM IR memory operations and their respective permission semantics.

The table outlines all the memory instructions as referenced by the LLVM Language Reference [51]. A few clarifications are in order:

- The observant reader might have noticed the missing `malloc` and `free` instructions for heap memory management. These were discontinued as of LLVM 2.7 [65] in favour of using the equivalent `malloc` and `free` instructions of the C standard library. Support for the C standard library is discussed in Section 9.4.
- Despite that the **`alloca`** instruction is only able to allocate memory on the stack, it should still grant permissions as well as require permission to access the memory. While the intended use case of stack memory is to store local variables, there are no mechanisms that prevent threads from accessing the memory stack of each other. For example, this can be achieved by a thread creating a pointer to its own stack memory and passing that pointer to the constructor for another thread. Therefore, stack memory is concurrently accessible and should therefore contain permissions.

[2]See: `https://releases.llvm.org/15.0.0/docs/AliasAnalysis.html`

- The `fence` instruction shares a lot of similarities with a *barrier* in parallel programming. Barriers are used in parallel programming to indicate a point in the program all involved threads should reach first before any thread is allowed to continue its execution. This checkpoint mechanism gives guarantees on certain computations having been completed and thus allowing for redistribution of permissions among the involved threads in the parallel program. A memory fence could be used for a similar use case and therefore shares some of its permission semantics with barriers.
- While `getelementptr` is strictly speaking a memory operation as it calculates memory pointer addresses, it only performs pointer arithmetics and thus does not interact with any memory. It is therefore completely free of any permission restrictions.

9.5.3 Memory Atomics

Concerning the complex matter that is atomic instruction reordering, VCLLVM will only be supporting atomic operations that are sequentially consistent. Instruction reorderings are advanced LLVM IR features and are currently a low priority for VCLLVM. However, support for non-atomic memory operations in a concurrent context can still be performed safely using other mechanisms such as locks and semaphores. These are part of the POSIX pthread library whose support is discussed in Section 9.4.

Acronyms

AST Abstract Syntax Tree.

CFG Control Flow Graph.

COL Common Object Language.

LLVM Low-Level Virtual Machine.

LLVM IR Low-Level Virtual Machine Intermediate Representation.

PBSL Permission-Based Separation Logic.

Protobuf Protocol Buffer.

PVL Prototypical Verification Language.

SSA Single Static Assignment.

VCLLVM VerCors Low-Level Virtual Machine.

Glossary

Φ Node Special control command in SSA code representation that merges divergent variable assignments of converging code branches.

Abstract Syntax Tree Graph representation of a program in the form of a tree that conveys the syntactical structure of the program.

Backend Compiler In the context of LLVM, a compiler that transforms LLVM IR into low-level machine code.

Common Object Language Internal code representation used inside VerCors and to which all AST transformations are applied.

Control Flow Graph Directed graph representing the control flow of a program with nodes as blocks of consecutive command sequences and edges as (conditional) branches.

Data Race Situation where two or more program threads simultaneously access the same memory location, potentially creating unpredictable program behaviour.

Deductive Verification Static verification technique that relies on a tool that generates mathematical proof obligations from code, annotations, and possibly other types of specifications to be provided to yet another verification tool to generate the proofs.

Frontend Compiler In the context of LLVM, a compiler that transforms a high-level source programming language into LLVM IR.

Hoare Logic Formal logic-based system to compositionally reason about the correctness of programs where each command in the program has a pre- and postcondition describing the state of the program before and after the command respectively (Section 2.1.2.1).

Interactive Theorem Proving Static verification technique that is based on a proof assistant tool that can automatically derive formal proofs but requires repeated human interaction to limit the search space for proofs.

LLVM IR Internal code representation that sits between high-level source code and the low-level machine code and is the target for code optimisation passes in the LLVM Project (Section 2.2.1).

Loop Invariant Predicate that must always hold right before the execution of a loop, after each execution of the loop body, and right after the execution of the entire loop.

Model Checking Static verification technique where a system is abstracted to a finite state machine and that is either exhaustively or boundedly checked if certain correctness properties and specifications hold.

Permission Essential construct in PBSL to indicate an ownership fraction p over a heap memory location where $0 < p < 1$ indicates a read permission and $p = 1$ indicates a write permission (Section 2.1.3.2).

Permission-Based Separation Logic Extension of Hoare logic to additionally be able to reason about heap memory usage through the notion of ownership, and read and write permissions (Section 2.1.2.2).

Postcondition Predicate that must always hold right after the execution of a command.

Precondition Predicate that must always hold right before the execution of a command.

Proof Obligation Logical formula associated with the correctness claim of a program.

Protocol Buffer Serialisation technology developed by Google to send compact binary serialisation of data between programs or interconnected systems.

Prototypical Verification Language Java-like toy programming language used in VerCors to model programs in.

Reducibility A CFG is said to be reducible if all intervals in the CFG are recursively reducible to a single interval (Section 4.4.1).

Single Static Assignment Code representation form that requires every variable to be assigned only once.

Specification Contract Code annotation used in deductive verification to describe mathematical properties about a certain piece of code.

Structured Programming Programming paradigm where programs are composed of concatenations (i.e. lines of codes being executed sequentially), selections (e.g. if-then-else structures), and repetitions (e.g. loop constructs like for- and while-loops) [15].

VCLLVM The external tool developed to work alongside VerCors to transform LLVM IR into a COL AST.

VCLLVM/VerCors The combination of the VCLLVM and VerCors tool together that makes end-to-end deductive verification of LLVM IR programs possible.

VerCors Deductive verification tool developed by the Formal Methods and Tools research group at the University of Twente based on permission-based separation logic (Section 2.1).

Viper Verification backend and verification language based on permission-based separation logic developed by the Programming Methodology Group of ETH Zürich [43].

Bibliography

[1] Frances E Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[2] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert Norton-Wright, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proceedings of the ACM on Programming Languages*, 2019.

[3] Alasdair Armstrong, Brian Campbell, Ben Simner, Christopher Pulte, and Peter Sewell. Isla: Integrating full-scale ISA semantics and axiomatic concurrency models. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I 33*, pages 303–316. Springer, 2021.

[4] Paolo Baldan, Roberto Bruni, Andrea Corradini, Fabio Gadducci, Hernan Melgratti, and Ugo Montanari. Event structures for petri nets with persistence. *arXiv preprint arXiv:1802.03726*, 2018.

[5] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

[6] Boehm Barry et al. Software engineering economics. *New York*, 197, 1981.

[7] Simon Bliudze, Petra van den Bos, Marieke Huisman, Robert Rubbens, and Larisa Safina. JavaBIP meets VerCors: Towards the safety of concurrent software systems in java. In Leen Lambers and Sebastián Uchitel, editors, *Fundamental Approaches to Software Engineering*, pages 143–150, Cham, 2023. Springer Nature Switzerland.

[8] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The VerCors tool set: verification of parallel and concurrent software. In *International Conference on Integrated Formal Methods*, pages 102–110. Springer, 2017.

[9] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, 2005.

[10] Guillaume Brat, Jorge A Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods: 12th International Conference, SEFM 2014, Grenoble, France, September 1-5, 2014. Proceedings 12*, pages 271–277. Springer, 2014.

[11] Soham Chakraborty and Viktor Vafeiadis. Formalizing the concurrency semantics of an LLVM fragment. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 100–110. IEEE, 2017.

[12] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[13] Dai Clegg and Richard Barker. *Case method fast-track: a RAD approach.* Addison-Wesley Longman Publishing Co., Inc., 1994.

[14] John Cocke and Raymond Miller. Some analysis techniques for optimizing computer programs. In *Proceedings of the Second Intl. Conf. of Systems Science*, 1969.

[15] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming.* Academic Press Ltd., 1972.

[16] Sharron Ann Danis. Rear Admiral Grace Murray Hopper, Feb 1997.

[17] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[18] Robert DeLine and K Rustan M Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical report, Citeseer, 2005.

[19] Edsger W Dijkstra. Cooperating sequential processes. In *The origin of concurrent programming*, pages 65–138. Springer, 1968.

[20] Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb. Constructing semantic models of programs with the software analysis workbench. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*, pages 56–72. Springer, 2016.

[21] Thomas Edison. Letter from Thomas Alva Edison to William Orton, Mar 1887.

[22] James Gosling, Bill Joy, Guy Lewis Steele Jr, Gilad Bracha, and Alex Buckley. The Java Language Specification: Java SE 8 Edition. Oracle America. *Inc., Redwood City, California, USA*, 2015.

[23] Lee L Gremillion. Determinants of program repair maintenance requirements. *Communications of the ACM*, 27(8):826–832, 1984.

[24] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361. Springer, 2015.

[25] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled C code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 637–651, 2020.

[26] Matthew S Hecht and Jeffrey D Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, 1972.

[27] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[28] IEC ISO. 9899: 2011 Information technology - Programming languages - C. Standard. *International Organization for Standardization, Geneva, Switzerland*, 2011.

[29] Capers Jones. *Software assessments, benchmarks, and best practices.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[30] T Capers Jones. *Estimating software costs.* McGraw-Hill, Inc., 1998.

[31] Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the ManySStuBs4J dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 573–577, 2020.

[32] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–32, 2017.

[33] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48:175–205, 2016.

[34] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++ 11. *ACM SIGPLAN Notices*, 52(6):618–632, 2017.

[35] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[36] K Rustan M Leino. This is Boogie 2. *manuscript KRML*, 178(131):9, 2008.

[37] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers 16*, pages 348–370. Springer, 2010.

[38] K Rustan M Leino, Peter Müller, and Jan Smans. Verification of concurrent programs with Chalice. In *FOSAD*, volume 5705, pages 195–222. Springer, 2009.

[39] Liyi Li and Elsa L Gunter. K-LLVM: a relatively complete semantics of LLVM IR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[40] Meri Lisi. Some remarks on the Cantor pairing function. *Le Matematiche*, 62(1):55–65, 2007.

[41] Didrik Lundberg, Roberto Guanciale, Andreas Lindner, and Mads Dam. Hoare-style logic for unstructured programs. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings*, pages 193–213. Springer, 2020.

[42] Steve McConnell. *Code complete.* Pearson Education, 2004.

[43] Peter Müller, Malte Schwerhoff, and Alexander J Summers. Viper: A verification infrastructure for permission-based reasoning. In *International conference on verification, model checking, and abstract interpretation*, pages 41–62. Springer, 2016.

[44] Luke Nelson, James Bornholt, Ronghui Gu, Andrew Baumann, Emina Torlak, and Xi Wang. Scaling symbolic evaluation for automated verification of systems code with Serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 225–242, 2019.

[45] The Editors of Encyclopædia Britannica. Fibonacci sequence, Jul 1998.

[46] Marco Paganoni and Carlo A Furia. Verifying functional correctness properties at the level of java bytecode. In *International Symposium on Formal Methods*, pages 343–363. Springer, 2023.

[47] Vaughan R Pratt. Semantical considerations on Floyd-Hoare logic. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 109–121. IEEE, 1976.

[48] LLVM Project. Exception handling in LLVM. `https://releases.llvm.org/15.0.0/docs/ExceptionHandling.html`, September 2022. [Accessed 19-Jan-2023].

[49] LLVM Project. LLVM atomic instructions and concurrency guide. `https://releases.llvm.org/15.0.0/docs/Atomics.html`, September 2022. [Accessed 19-Jan-2023].

[50] LLVM Project. LLVM developer policy: IR backwards compatibility. `https://llvm.org/docs/DeveloperPolicy.html#ir-backwards-compatibility`, December 2022. [Accessed 05-Dec-2022].

[51] LLVM Project. LLVM language reference manual. `https://releases.llvm.org/15.0.0/docs/LangRef.html`, September 2022. [Accessed 05-Dec-2022].

[52] LLVM Project. LLVM loop terminology (and canonical forms). `https://releases.llvm.org/15.0.0/docs/LoopTerminology.html`, September 2022. [Accessed 12-Jun-2023].

[53] LLVM Project. opt - LLVM optimizer. `https://releases.llvm.org/15.0.0/docs/CommandGuide/opt.html`, September 2022. [Accessed 05-Dec-2022].

[54] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 106–113. Springer, 2014.

[55] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002.

[56] Heinz Riener and Görschwin Fey. FAuST: A framework for formal verification, automated debugging, and software test generation. In *Model Checking Software: 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings 19*, pages 234–240. Springer, 2012.

[57] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong-execution of LLVM-based languages on the JVM: Position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 1–4, 2016.

[58] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[59] Michael Sammler, Angus Hammond, Rodolphe Lepigre, Brian Campbell, Jean Pichon-Pharabod, Derek Dreyer, Deepak Garg, and Peter Sewell. Islaris: verification of machine code against authoritative ISA semantics. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 825–840, 2022.

[60] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, volume 16, 2013.

[61] Micha Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages*, 5(3-4):141–153, 1980.

[62] NJA Sloane and Simon Plouffe. The encyclopedia of integer sequences academic press. *San Diego, CA*, 1995.

[63] Marcelo Sousa and Alper Sen. LLVMVF: A generic approach for verification of multicore software. *Journal of Electronic Testing*, 29:635–646, 2013.

[64] Matthew Szudzik. An elegant pairing function. In *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*, pages 1–12, 2006.

[65] The LLVM Team. LLVM 2.7 release notes, April 2010.

[66] The LLVM Team. LLVM 3.1 release notes, May 2012.

[67] The LLVM Team. LLVM 12.0.1 release notes, July 2021.

[68] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[69] Freark I van der Berg. LLMC: Verifying high-performance software. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part II 33*, pages 690–703. Springer, 2021.

[70] Freark I van der Berg. Recursive variable-length state compression for multi-core software model checking. In *NASA Formal Methods: 13th International Symposium, NFM 2021, Virtual Event, May 24–28, 2021, Proceedings*, pages 340–357. Springer, 2021.

[71] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One VM to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 187–204, 2013.

[72] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in Coq. *arXiv preprint arXiv:1906.00046*, 2019.

[73] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. Citeseer, 2015.

[74] Edward Yourdon. *Modern structured analysis.* Yourdon press, 1989.

[75] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *Proceedings of the ACM on Programming Languages*, 5(ICFP):1–30, 2021.

[76] Vadim Zaliva and Franz Franchetti. HELIX: a case study of a formal verification of high performance program generation. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 1–9, 2018.

[77] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 427–440, 2012.

Appendix A

Specification Expression Grammar

N.B. Explanation on target independent specification language syntax can be found here: `https://github.com/utwente-fmt/vercors/wiki/Specification-Syntax`

⟨*expression*⟩ ::= ⟨*instruction*⟩
| ⟨*constant*⟩
| ⟨*identifier*⟩
| ⟨*expression*⟩ '==>' ⟨*expression*⟩
| ⟨*valExpr*⟩ *inherited rule from the global spec. grammar in VerCors*

⟨*instruction*⟩ ::= ⟨*binOpInstruction*⟩
| ⟨*compareInstruction*⟩
| ⟨*callInstruction*⟩
| ⟨*branchInstruction*⟩

⟨*binOpInstruction*⟩ ::= ⟨*binOp*⟩ '(' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'

⟨*binOp*⟩ ::= 'add' | 'sub' | 'mul' | 'udiv' | 'sdiv' | 'and' | 'or' | 'xor'

⟨*compareInstruction*⟩ ::= ⟨*compOp*⟩ '(' ⟨*compPred*⟩ ',' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'

⟨*compOp*⟩ ::= 'icmp' *currently only integer comparison is supported*

⟨*compPred*⟩ ::= 'eq' | 'ne' | 'ugt' | 'uge' | 'ult' | 'ule' | 'sgt' | 'sge' | 'slt' | 'sle'

⟨*callInstruction*⟩ ::= 'call' ⟨*identifier*⟩ '(' ⟨*expressionList*⟩ ')'

⟨*expressionList*⟩ ::= ⟨*expression*⟩ | ⟨*expression*⟩ ',' ⟨*expressionList*⟩

⟨*branchInstruction*⟩ ::= 'br (' ⟨*expression*⟩ ',' ⟨*expression*⟩ ',' ⟨*expression*⟩ ')'

⟨*constant*⟩ ::= 'true' | 'false' | '-'?[0-9]+

⟨*identifier*⟩ ::= [%@][-a-zA-Z$._][-a-zA-Z$._0-9]* | [%@][0-9]+

Appendix B

Additional Verification Examples

In addition to the verification examples presented in Section 7.2, this appendix aims to expand on this. The examples in this appendix are divided into four categories: Types (Section B.1), Arithmetics (Section B.2), Branching (Section B.3), and Functions (Section B.4).

The examples have been prepared using the methodology described in Chapter 7. For each example, the original C program is first presented followed by a compiled version of the program in LLVM IR. Specifications that verify are marked in green while specifications that do not verify are marked in red.

B.1 Types

B.1.1 Integers

```
long long_const() {
    return 420;
}

long long_var(long n) {
    return n;
}

int int_const() {
    return 69;
}

int int_var(int n) {
    return n;
}

short short_const() {
    return 42;
}

short short_var(short n) {
    return n;
}

uint32_t unsigned_const() {
    return 1337;
}

uint32_t unsigned_var(uint32_t n) {
    return n;
}

int64_t signed_const() {
    return -0xBEEF;
}

int64_t signed_var(int64_t n) {
    return n;
}

// unsupported: type mixing in argument vs return:
// (e.g. arg:short -> ret:long)
```

```
define dso_local i64 @long_const()
!VC.contract !{
!"ensures icmp(eq, \result, 420);"
!"ensures icmp(eq, \result, 0);"
}
{
  ret i64 420
}

define dso_local i64 @long_var(i64 noundef %0)
!VC.contract !{
!"ensures icmp(eq, \result, %0);"
!"ensures icmp(ne, \result, %0);"
}
{
  ret i64 %0
}

define dso_local i32 @int_const()
!VC.contract !{
!"ensures icmp(eq, \result, 69);"
!"ensures icmp(eq, \result, 0);"
}
{
  ret i32 69
}

define dso_local i32 @int_var(i32 noundef %0)
!VC.contract !{
!"ensures icmp(eq, \result, %0);"
!"ensures icmp(ne, \result, %0);"
}
{
  ret i32 %0
}

define dso_local signext i16 @short_const()
!VC.contract !{
!"ensures icmp(eq, \result, 42);"
!"ensures icmp(eq, \result, 0);"
}
{
  ret i16 42
}
```

```
45 define dso_local signext i16 @short_var(i16 noundef signext %0)
46 !VC.contract !{
47 !"ensures icmp(eq, \result, %0);"
48 !"ensures icmp(ne, \result, %0);"
49 }
50 {
51   ret i16 %0
52 }
53
54 define dso_local i32 @unsigned_const()
55 !VC.contract !{
56 !"ensures icmp(eq, \result, 1337);"
57 !"ensures icmp(eq, \result, 0);"
58 }
59 {
60   ret i32 1337
61 }
62
63 define dso_local i32 @unsigned_var(i32 noundef %0)
64 !VC.contract !{
65 !"ensures icmp(eq, \result, %0);"
66 !"ensures icmp(ne, \result, %0);"
67 }
68 {
69   ret i32 %0
70 }
71
72 define dso_local i64 @signed_const()
73 !VC.contract !{
74 !"ensures icmp(eq, \result, -48879);"
75 !"ensures icmp(eq, \result, 0);"
76 }
77 {
78   ret i64 -48879
79 }
80
81 define dso_local i64 @signed_var(i64 noundef %0)
82 !VC.contract !{
83 !"ensures icmp(eq, \result, %0);"
84 !"ensures icmp(ne, \result, %0);"
85 }
86 {
87   ret i64 %0
88 }
```

B.1.2 Booleans

```c
#include "stdbool.h"

bool const_true() {
    return true;
}

bool const_false() {
    return false;
}

// unsupported: variable boolean return (causes a truncate on return)
```

```llvm
define dso_local zeroext i1 @const_true()
!VC.contract !{
!"ensures icmp(eq, \result, true);"
!"ensures icmp(eq, \result, false);"
}
{
  ret i1 true
}


define dso_local zeroext i1 @const_false()
!VC.contract !{
!"ensures icmp(eq, \result, false);"
!"ensures icmp(eq, \result, true);"
}
{
  ret i1 false
}
```

B.2 Arithmetics

B.2.1 Addition, Subtraction, and Multiplication

```
1  int add(int x, int y) {
2      return x + y;
3  }
4
5  int sub(int x, int y) {
6      return x - y;
7  }
8
9  int mul(int x, int y) {
10     return x * y;
11 }
```

```
1  define dso_local i32 @add(i32 noundef %0, i32 noundef %1)
2  !VC.contract !{
3  !"ensures icmp(eq, \result, add(%0, %1));"
4  !"ensures icmp(eq, \result, add(%0, %0));"
5  }
6  {
7    %3 = add nsw i32 %0, %1
8    ret i32 %3
9  }
10
11 define dso_local i32 @sub(i32 noundef %0, i32 noundef %1)
12 !VC.contract !{
13 !"ensures icmp(eq, \result, sub(%0, %1));"
14 !"ensures icmp(eq, \result, sub(%0, %0));"
15 }
16 {
17   %3 = sub nsw i32 %0, %1
18   ret i32 %3
19 }
20
21 define dso_local i32 @mul(i32 noundef %0, i32 noundef %1)
22 !VC.contract !{
23 !"ensures icmp(eq, \result, mul(%0, %1));"
24 !"ensures icmp(eq, \result, mul(%0, %0));"
25 }
26 {
27   %3 = mul nsw i32 %0, %1
28   ret i32 %3
29 }
```

B.2.2 (Un)signed Division

```
1  #include "stdint.h"
2
3  uint32_t unsignedDiv(uint32_t x, uint32_t y) {
4      return x / y;
5  }
6
7  int32_t signedDiv(int32_t x, int32_t y) {
8      return x / y;
9  }
10
11 int32_t signedByUnsigned(int32_t x, uint32_t y) {
12     return x / y;
13 }
14
15 int32_t unsignedBySigned(uint32_t x, int32_t y) {
16     return x / y;
17 }
```

```
define dso_local i32 @unsignedDiv(i32 noundef %0, i32 noundef %1)
!VC.contract !{
!"requires icmp(ne, %1, 0);",
!"ensures icmp(eq, \result, udiv(%0, %1));"
!"ensures icmp(eq, \result, 0);"
}
{
  %3 = udiv i32 %0, %1
  ret i32 %3
}

define dso_local i32 @signedDiv(i32 noundef %0, i32 noundef %1)
!VC.contract !{
!"requires icmp(ne, %1, 0);",
!"ensures icmp(eq, \result, sdiv(%0, %1));"
!"ensures icmp(eq, \result, 0);"
}
{
  %3 = sdiv i32 %0, %1
  ret i32 %3
}

define dso_local i32 @signedByUnsigned(i32 noundef %0, i32 noundef %1)
!VC.contract !{
!"requires icmp(ne, %1, 0);",
!"ensures icmp(eq, \result, udiv(%0, %1));"
!"ensures icmp(eq, \result, 0);"
}
{
  %3 = udiv i32 %0, %1
  ret i32 %3
}

define dso_local i32 @unsignedBySigned(i32 noundef %0, i32 noundef %1)
!VC.contract !{
!"requires icmp(ne, %1, 0);",
!"ensures icmp(eq, \result, udiv(%0, %1));"
!"ensures icmp(eq, \result, 0);"
}
{
  %3 = udiv i32 %0, %1
  ret i32 %3
}
```

B.3 Branching

B.3.1 If-then

```
1  int if_then_multi_ret(int x) {
2      if(x > 0) {
3          return 1;
4      }
5      return 0;
6  }
7
8  int if_then_single_ret(int x) {
9      int i = 0;
10     if(x > 0) {
11         i = 1;
12     }
13     return i;
14 }
```

```
1  define dso_local i32 @if_then_multi_ret(i32 noundef %0)
2  !VC.contract !{
3  !"ensures icmp(eq, \result, br(icmp(sgt, %0, 0), 1, 0));"
4  !"ensures icmp(eq, \result, 0);"
5  }
6  {
7    %2 = icmp sgt i32 %0, 0
8    br i1 %2, label %3, label %4
9  3:
10   br label %5
11 4:
12   br label %5
13 5:
14   %.0 = phi i32 [ 1, %3 ], [ 0, %4 ]
15   ret i32 %.0
16 }
17
18 define dso_local i32 @if_then_single_ret(i32 noundef %0)
19 !VC.contract !{
20 !"ensures icmp(eq, \result, br(icmp(sgt, %0, 0), 1, 0));"
21 !"ensures icmp(eq, \result, 0);"
22 }
23 {
24   %2 = icmp sgt i32 %0, 0
25   br i1 %2, label %3, label %4
26 3:
27   br label %4
28 4:
29   %.0 = phi i32 [ 1, %3 ], [ 0, %1 ]
30   ret i32 %.0
31 }
```

B.3.2 If-then-else

```c
#include "stdbool.h"

int if_then_else_multi_ret(int x) {
    if(x != 0) {
        return 1;
    } else {
        return 0;
    }
}

int if_then_else_single_ret(int x) {
    int i;
    if(x != 0) {
        i = 1;
    } else {
        i = 0;
    }
    return i;
}
```

```
define dso_local i32 @if_then_else_multi_ret(i32 noundef %0)
!VC.contract !{
!"ensures icmp(eq, \result, br(icmp(ne, %0, 0), 1, 0));"
!"ensures icmp(eq, \result, 0);"
}
{
  %2 = icmp ne i32 %0, 0
  br i1 %2, label %3, label %4
3:
  br label %5
4:
  br label %5
5:
  %.0 = phi i32 [ 1, %3 ], [ 0, %4 ]
  ret i32 %.0
}

define dso_local i32 @if_then_else_single_ret(i32 noundef %0)
define dso_local i32 @if_then_else_multi_ret(i32 noundef %0)
!VC.contract !{
!"ensures icmp(eq, \result, br(icmp(ne, %0, 0), 1, 0));"
!"ensures icmp(eq, \result, 0);"
}
{
  %2 = icmp ne i32 %0, 0
  br i1 %2, label %3, label %4
3:
  br label %5
4:
  br label %5
5:
  %.0 = phi i32 [ 1, %3 ], [ 0, %4 ]
  ret i32 %.0
}
```

B.3.3 Nested Branches

```
1  int branch_in_branch_multi_ret(int x, int y) {
2      if(x > 0) {
3          if(y > 0) {
4              return 2;
5          } else {
6              return 1;
7          }
8      } else {
9          return 0;
10     }
11 }
12
13 int branch_in_branch_single_ret(int x, int y) {
14     int i;
15     if(x > 0) {
16         if(y > 0) {
17             i = 2;
18         } else {
19             i = 1;
20         }
21     } else {
22         i = 0;
23     }
24     return i;
25 }
```

```
1  define dso_local i32 @branch_in_branch_multi_ret(i32 %0, i32 %1)
2  !VC.contract !{
3  !"ensures icmp(eq, \result, br(icmp(sgt, %0, 0),
4                                  br(icmp(sgt, %1, 0), 2, 1),
5                                  0));"
6  !"ensures icmp(eq, \result, 0);"
7  }
8  {
9    %3 = icmp sgt i32 %0, 0
10   br i1 %3, label %4, label %8
11 4:
12   %5 = icmp sgt i32 %1, 0
13   br i1 %5, label %6, label %7
14 6:
15   br label %9
16 7:
17   br label %9
18 8:
19   br label %9
20 9:
21   %.0 = phi i32 [ 2, %6 ], [ 1, %7 ], [ 0, %8 ]
22   ret i32 %.0
23 }
24
25 define dso_local i32 @branch_in_branch_single_ret(i32 %0, i32 %1)
26 !VC.contract !{
27 !"ensures icmp(eq, \result, br(icmp(sgt, %0, 0),
28                                 br(icmp(sgt, %1, 0), 2, 1),
29                                 0));"
30 !"ensures icmp(eq, \result, 0);"
31 }
32 {
33   %3 = icmp sgt i32 %0, 0
34   br i1 %3, label %4, label %9
35 4:
36   %5 = icmp sgt i32 %1, 0
37   br i1 %5, label %6, label %7
38 6:
39   br label %8
40 7:
41   br label %8
42 8:
43   %.0 = phi i32 [ 2, %6 ], [ 1, %7 ]
44   br label %10
45 9:
46   br label %10
47 10:
48   %.1 = phi i32 [ %.0, %8 ], [ 0, %9 ]
49   ret i32 %.1
50 }
```

B.4 Functions

B.4.1 Function Call

```
1  int square(int x) {
2      return x * x;
3  }
4
5  int square_call(int x) {
6      return foo(x);
7  }
```

```
1  define dso_local i32 @square(i32 noundef %0)
2  !VC.pure !{i1 true}
3  {
4    %2 = mul nsw i32 %0, %0
5    ret i32 %2
6  }
7
8
9  define dso_local i32 @bar(i32 noundef %0)
10 !VC.contract !{
11 !"ensures icmp(eq, \result, mul(%0, %0));",
12 !"ensures icmp(eq, \result, call @square(%0));"
13 !"ensures icmp(eq, \result, 0);",
14 }
15 {
16   %2 = call i32 @square_call(i32 noundef %0)
17   ret i32 %2
18 }
```

B.4.2 Cyclic Function Call

```
1  int bar(int x);
2
3  int foo(int x) {
4      if(x <= 0) {
5          return 1;
6      }
7      return bar(x - 1);
8  }
9
10 int bar(int x) {
11     if(x <= 0) {
12         return 0;
13     }
14     return foo(x - 1);
15 }
```

```
define dso_local i32 @foo(i32 noundef %0)
!VC.contract !{
!"ensures or(icmp(eq, \result, 0), icmp(eq, \result, 1));"
!"ensures icmp(eq, \result, 0);
}
{
  %2 = icmp sle i32 %0, 0
  br i1 %2, label %3, label %4
3:
  br label %7
4:
  %5 = sub nsw i32 %0, 1
  %6 = call i32 @bar(i32 noundef %5)
  br label %7
7:
  %.0 = phi i32 [ 1, %3 ], [ %6, %4 ]
  ret i32 %.0
}

define dso_local i32 @bar(i32 noundef %0)
!VC.contract !{
!"ensures or(icmp(eq, \result, 0), icmp(eq, \result, 1));"
!"ensures icmp(eq, \result, 0);
}
{
  %2 = icmp sle i32 %0, 0
  br i1 %2, label %3, label %4
3:
  br label %7
4:
  %5 = sub nsw i32 %0, 1
  %6 = call i32 @foo(i32 noundef %5)
  br label %7
7:
  %.0 = phi i32 [ 0, %3 ], [ %6, %4 ]
  ret i32 %.0
}
```

B.4.3 Multiplication with Recursive Summing

```
int recursive_mult(int x, int y) {
    if (x > 0) {
        return y + recursive_mult(x - 1, y);
    }
    return 0;
}
```

```
define dso_local i32 @recursive_mult(i32 noundef %0, i32 noundef %1)
!VC.contract !{
!"requires icmp(sge, %0, 0);",
!"requires icmp(sge, %1, 0);",
!"ensures icmp(sge, \result, mul(%0, %1));"
!"ensures icmp(sgt, \result, 0);"
}
{
  %3 = icmp sgt i32 %0, 0
  br i1 %3, label %4, label %8
4:
  %5 = sub nsw i32 %0, 1
  %6 = call i32 @recursive_mult(i32 noundef %5, i32 noundef %1)
  %7 = add nsw i32 %1, %6
  br label %9
8:
  br label %9
9:
  %.0 = phi i32 [ %7, %4 ], [ 0, %8 ]
  ret i32 %.0
}
```

B.4.4 Factorial

```
int factorial(int x) {
    if(x > 1) {
        return x * factorial(x - 1);
    }
    return 1;
}
```

```
1  !VC.global = !{!0}
2  !0 = !{
3  !"pure i32 @fact(i32 %n) = br(icmp(sgt, %n, 1),
4                              mul(call @fact(sub(%n, 1)), %n),
5                              1);"
6  }
7
8  define dso_local i32 @factorial(i32 noundef %0)
9  !VC.contract !{
10 !"requires icmp(sge, %0, 1);",
11 !"ensures icmp(eq, \result, call @fact(%0));"
12 !"ensures icmp(ne, \result, call @fact(%0));"
13 }
14 {
15   %2 = icmp sgt i32 %0, 1
16   br i1 %2, label %3, label %7
17 3:
18   %4 = sub nsw i32 %0, 1
19   %5 = call i32 @factorial(i32 noundef %4)
20   %6 = mul nsw i32 %0, %5
21   br label %8
22 7:
23   br label %8
24 8:
25   %.0 = phi i32 [ %6, %3 ], [ 1, %7 ]
26   ret i32 %.0
27 }
```