First Steps towards Deductive Verification of LLVM IR [⋆]

Dré van Oorschot, Marieke Huisman(✉), and Ömer Şakar

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
d.h.m.a.vanoorschot@alumnus.utwente.nl,
{m.huisman,o.f.o.sakar}@utwente.nl

Abstract. Over the last years, deductive program verifiers have substantially improved, and their applicability on non-trivial applications has been demonstrated. However, a major bottleneck is that for every new programming language, a new deductive verifier has to be built.
This paper describes the first steps in a project that aims to address this problem, by language-agnostic support for deductive verification: Rather than building a deductive program verifier for every programming language, we develop deductive program verification technology for a widely-used intermediate representation language (LLVM IR), such that we eventually get verification support for any language that can be compiled into the LLVM IR format.
Concretely, this paper describes the design of VCLLVM, a prototype tool that adds LLVM IR as a supported language to the VerCors verifier. We discuss the challenges that have to be addressed to develop verification support for such a low-level language. Moreover, we also sketch how we envisage to build verification support for any specified source program that can be compiled into LLVM IR on top of VCLLVM.

1 Introduction

As software has become an intrinsic part of our daily lives, we become more and more dependent on software being reliable and dependable, and we need tools that can help us to establish these guarantees. Over the last years, substantial progress has been made in the development of *formal verification techniques* that can be used to ensure that software provides certain guarantees. This covers a wide range of different approaches that can be used to provide guarantees at different levels of abstraction and precision. Here, we focus in particular on *deductive program verification* techniques [11], which are used to provide guarantees directly at code level, by verifying whether a program fragment behaves according to the pre-postcondition-contract that is specified for it. A broad range of deductive verifiers exist, such as VerCors [4], KeY [1], VeriFast [14, 15], Viper [25], Dafny [20], RESOLVE [37], Whiley [31], Frama-C [3], KIV [9] and OpenJML [7], which have been used in several non-trivial case studies, see e.g. [29, 35, 34, 29,

⋆ Work on this project is supported by the NWO VICI 639.023.710 Mercedes project and the NWO TTW 17249 ChEOPS project.

13, 10, 17]. A major challenge for deductive verifiers in practice is to enlarge the particular language features that they support. This language-dependency creates a severe limitation on how effective these techniques can be used in current software development, where language standards are regularly updated, new programming languages are frequently used, and applications are often written using multiple programming languages.

In compiler technology, this growth in source level programming languages, as well as the wide range of target architectures has been tackled by the introduction of intermediate representation formats, such as LLVM IR [19]. They require only a compiler into this intermediate representation format for a new programming language, while new architectures are supported by defining a mapping from the intermediate representation format into the new hardware. We propose a similar approach to reduce the language-dependence of deductive program verification technology, by: (1) defining verification technology for LLVM IR, and (2) developing a generic approach to translate contract specifications from a wide range of source languages into contract specifications for LLVM IR.

This paper focuses in particular on the first step in this project: it contributes VCLLVM, a prototype tool that encodes annotated LLVM IR programs into the VerCors verifier [4] to enable deductive verification for LLVM IR. We describe the challenges for the encoding of LLVM IR into VerCors, as LLVM IR is a much lower-level language than the languages that are supported by VerCors already, and how these challenges affect the design and implementation of VCLLVM. We also sketch how we plan to use VCLLVM as a stepping stone in a bigger project to develop language-independent support for deductive verification.

2 Background

This section gives a brief background on the VerCors verifier and LLVM IR.

VerCors VerCors [4] is a deductive verifier for concurrent programs. It can verify programs written in several programming languages (e.g., Java, CUDA, OpenCL, and its internal Prototype Verification Language PVL). To verify programs with VerCors, they are first annotated with pre-postcondition-contract specifications written in *permission-based separation logic* (PBSL) [38], and then the specified programs are encoded into the internal format of VerCors, called COL, which is transformed in several steps into the input language of Viper [25]. The Viper infrastructure is then used for verification. If verification with Viper fails, VerCors translates the error message back to the level of the source program.

PBSL is a concurrent separation logic [27] with support for permissions [5]. Permissions make the language suitable to reason about concurrent programs, as they are used to encode when variables may be read or written. VCLLVM at the moment only supports sequential programs, thus we do not provide further details about PBSL here, and instead refer to the documentation.

LLVM IR LLVM IR (LLVM Intermediate Representation) is the common interface for the *frontend* and *backend* compilers developed as part of the LLVM project [19]. LLVM IR is designed to be *abstract* enough to be compiled to from higher level frontend languages, and *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 [23]. More details about LLVM IR can be found in its documentation [22].

The LLVM IR language is an assembly language using the single static assignment format. Each LLVM IR file consists of one module. Each module contains multiple functions. Functions are divided into multiple (possibly labelled) blocks, with one dedicated entry block. Every block consists of one or more instructions. We briefly summarise the main features of LLVM IR that are relevant for our work. First of all, LLVM IR features only two basic types, namely integers and floats, with the standard (bitwise) binary operators. Both come with different precisions. These two basic types can be combined into aggregate types, such as vectors, arrays, and structs, and can be referenced via pointers. Further, LLVM IR supports custom-declared constants and several predefined constants, such as **`true`** and **`false`**. The constant `undef` is used to present undefined state to the compiler as a range of possible values, which guarantees that the program itself remains well-defined. The constant `poison` indicates erroneous state of a program. LLVM IR offers branch instructions that can conditionally jump to the beginning of any instruction block in the same function. This can be used to encode conditionals and loops, and it offers a basis for error handling instructions. It is important to note that the internals of LLVM IR are not stable, meaning there are no guarantees for compatibility between different LLVM IR versions [21]. However, there are stable LLVM API functions that can analyse and manipulate the internals of LLVM IR.

3 Challenges for Deductive Verification of LLVM IR

In order to encode LLVM IR programs into input for the VerCors verifier, several challenges need to be addressed, as discussed in this section. The next section discusses how these challenges influence our prototype design and implementation. In particular, challenge 1 to 3 have been addressed in our prototype, while providing full solutions to challenges 4 to 7 has been left as future work.

– *Challenge 1: Instability of LLVM IR* As mentioned, LLVM IR is an unstable language [21], without backwards compatibility, and there is no guaranteed interoperability between the syntax of LLVM IR of different LLVM versions.
– *Challenge 2: LLVM IR Specifications* VerCors specifications use expressions from the source language. As expressions in LLVM IR are written as a block of single instructions, this raises the question what a suitable specification language for LLVM IR would be: writing blocks in specifications (or even multiple blocks with branches, e.g. for Boolean expressions) would be impractical and error-prone. However, an upside is that LLVM IR uses the SSA (static single-assignment) format, which makes it hard to write specifications that have side effects, and all instructions in LLVM IR are pure except for memory instructions such as `store` and `alloca`.

- *Challenge 3: Origin of User Errors* Parsing an LLVM IR file with the parser of the LLVM API returns a module object that does not retain any origin information; it is merely a semantically equivalent in-memory representation of the program. This makes it challenging to communicate the origin of a verification problem in the source code to the user. LLVM offers the possibility to construct a string of LLVM IR representing any LLVM value, but calculating line and column numbers or extracting a source string is complex as extraneous white spaces and comments in the source file are ignored.
- *Challenge 4: Control Flow* 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 be verified. VerCors technically supports goto statements but there are some caveats to be aware of when using them: the inclusion of goto statements obstructs the guarantee that the program is reducible [12], and loop invariants are hard to verify when a loop contains arbitrary goto statements.

  With that in mind, the encoding essentially needs to be an LLVM IR decompiler to the high-level COL representation of VerCors. Loops can be especially hard to recover due to their various forms (e.g. for-loops, while-loops, and do-while loops), and the possibility of nesting. The challenge is not so much in detecting cycles in the CFG (control flow graph) of the program (for which trivial graph algorithms exist), but mainly to identify the different parts of the loop (e.g., the loop condition, the loop body, and loop breaks).
- *Challenge 5: Low-level Language Features* LLVM IR introduces new low-level language constructs that have not been handled by VerCors yet, such as loads, stores and other low-level memory instructions, Φ nodes (from the SSA format), and low-level exception handling. All these concepts have to be integrated into COL.

  The current VCLLVM prototype simplifies many of these concepts or has not yet implemented them. Some ideas on how other LLVM IR low-level concepts could be translated into COL are discussed in [28].
- *Challenge 6: LLVM Concurrency Model* While LLVM IR does support instructions and control mechanisms that can be useful to ensure thread safety, it does not support constructs for parallel thread creation or signal handling natively. Instead, LLVM IR code depends on being linked against existing concurrency libraries, e.g. the pthread library on POSIX systems for Clang. Thus, in order to support reasoning about these concurrency libraries, their behaviour has to be modeled.
- *Challenge 7:* `undef` *and* `poison` Both constants `undef` and `poison` are semantically complex, and it is challenging to capture their semantics into VerCors. First, `undef` represents a set of possible values, which should be semantically treated as if it is a single value, and this concept does not yet exist in VerCors. Second, `poison` indicates erroneous behaviour, and it will have to be integrated into exception handling support of VerCors.

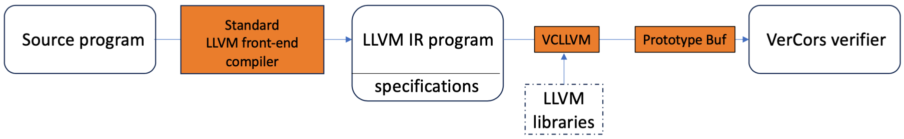

Fig. 1: Workflow of using VCLLVM and VerCors

4 Design and Implementation of VCLLVM

This section discusses the design and implementation of our prototype tool VCLLVM that translates LLVM IR programs into the VerCors internal COL format. Figure 1 gives a general overview how VCLLVM connects to VerCors. We discuss the main decisions in the design of VCLLVM, taking into account the challenges mentioned above. For a more in-depth analysis of the design choices, we refer to the Master thesis accompanying this paper [28].

Embedding versus Externalising The first design choice was whether to embed VCLLVM into the VerCors codebase or to develop it as an extension. Embedding could exacerbate the problems of Challenge 1 (instability of LLVM IR), and it would also restrict the tool implementation language to be JVM-compatible, which makes it hard to interface with existing LLVM IR functionality from the LLVM project. Instead, externalising makes it possible to use C++ to implement VCLLVM and to use all existing LLVM support functionality. We decided to go for this option, as it makes VCLLVM easier to maintain in the future.

VCLLVM Output Format As VCLLVM is developed as an external tool, its output needs to be in a format that is either already interpretable by VerCors or for which an interpreter would be simple to implement. If VCLLVM would generate concrete syntax, this requires that we define a concrete input language that supports all features of LLVM IR. Instead, we opted to use serialisation, which makes it possible to connect to the internal COL AST directly. We use *Protocol Buffers*[1] for this. It offers a largely automatable serialisation method, with language support for Scala (implementation language of VerCors) and C++ (implementation language of VCLLVM). Moreover, it supports code generation both from and to a Protocol Buffer definition, which simplifies the development of the communication layer between VCLLVM and VerCors considerably.

Specification Syntax To specify the properties that need to be verified, we need to embed 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 LLVM parser, the only option available is to use LLVM metadata to embed specifications.

[1] See: `https://developers.google.com/protocol-buffers`.

```
!VC.contract !{
   !"ensures",
   !"%var1 = mul i32 %y, %x",
   !"%var2 = add i32 %var1, %z",
   !"%verdict = icmp eq i32 %var2, \result;"
 }
```

(a)

```
!VC.contract !{
   !"ensures %x * %y + %z == \result;"
 }
```

(b)

```
!VC.contract !{
  !"ensures icmp(eq,
      add(mul(%y, %x), %z), \result);"
  }
```

(c)

Fig. 2: Possible Specification Syntax Options

Ideally, the specification syntax stays as close as possible to the LLVM IR syntax, but as explained in Challenge 2, it is not obvious for LLVM IR because of its low-level nature. We considered 3 different options, as illustrated in Figure 2 with contracts that describe the following *add-multiply* LLVM IR function.

```
define i32 @addMult(i32 %x, i32 %y, i32 %z)
!VC.contract !1 ;, !2 or !3 from Figure 2  {
  %1 = mul i32 %y, %x
  %res2 = add i32 %1, %z
  ret i32 %res2 }
```

This function takes as input parameters `x`, `y` and `z`. First it multiplies `x` and `y`, stores the intermediate result in a local variable `%1`, and then adds `z` to this, and returns this final result. All specifications in Figure 2 express that the return value is equal to `x * y + z`. As usual, we use the keyword `ensures` to specify a postcondition of the function, and `\result` to refer to the output value of the function. Figure 2a uses blocks of instructions to write the specification expressions. This is verbose, error prone and complicates parsing. Figure 2b uses a specification syntax that is independent of LLVM IR syntax. This is readable, but also creates ambiguities, as it makes it harder to connect the specification to the code. Finally, Figure 2c uses the known LLVM IR instruction keywords, but in a more functional manner. This is fairly readable, and avoids the ambiguity. We decided to use this option for VCLLVM. Notice that, as described in Section 7, eventually we hope to use VCLLVM as an intermediate tool to reason about programs in any language that compiles into LLVM IR. In that set up, the specification would be written in the input language of the high-level language, and compiled into a VCLLVM specification.

External library support LLVM IR is often compiled and linked against existing libraries to provide support for external libraries. Support for this is needed in particular to reason about concurrent LLVM IR, which rely on thread libraries. The VCLLVM prototype has been designed with this requirement in mind, but it has not yet been implemented.

5 Evaluation

To use the current version of VCLLVM, one needs to (1) write C code, (2) compile that C code to LLVM IR, (3) optionally run the LLVM opt tool [23] to mitigate program structures VCLLVM cannot yet interpret, (4) annotate the resulting LLVM IR program manually, and (5) 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, the regression test suite of VCLLVM currently only supports

The tool is only a prototype, but it has been used on several non-trivial examples, such as functions to compute triangular numbers and Cantor pairs, a function for date comparison (using branching and integer comparison), and recursive functions like Fibonacci and the factorial. In order to specify functional behaviour of these programs, VCLLVM supports the definition of pure specification-only functions, such as for example `fib`:

```
!VC.global = !{!0}
!0 = !{
!"pure i32 @fib(i32 %n) =
  br(icmp(sgt, %n, 2),
    add(call @fib(sub(%n, 1)), call @fib(sub(%n, 2))),1);”}
```

This expresses that for any `fib(n)` is computed using the following expression: `if(n > 2) then fib(n - 1) + fib(n - 2) else 1` (where `br` denotes a branch and `icmp` compares two integers).

Using this function, we can write and prove the following contract for a recursive implementation of the Fibonacci function, see [28] for the full program. This contract states that for any `n > 1`, the correct Fibonacci value is returned.

```
define dso_local i32 @fibonacci(i32 noundef %0)
!VC.contract !{
!"requires icmp(sge, %0, 1);",
!"ensures icmp(eq, \result, call @fib(%0));"
}
{ ... }
```

Special attention has been given to give informative feedback when verification fails. For more details about these examples, we refer to [28].

6 Related Work

There exist several projects that develop formal static analysis techniques for bug finding in LLVM IR. SMACK [32] defines a translation of LLVM IR into BoogiePL [20], to reason about C-programs using assertions that are compiled into LLVM IR using Clang. The verification itself is bounded and a potential extension to contract specifications has not yet been explored. The Vellvm project [40, 39]) develops a framework to reason about LLVM IR programs. It provides a mechanised semantics for LLVM IR, which can be used for verification. Reasoning is done directly in Coq, rather than at the code level, which requires Coq expertise. KLEE [6]. is a dynamic symbolic execution engine, which automatically generates suitable unit tests for LLVM IR applications, with a much better coverage than manually created test suites, thus increasing the likelihood of finding bugs. However, KLEE focuses only on bug finding, not on proving correctness. Another recent tool to easily find bugs via a bounded analysis of LLVM IR programs is Alive2 [24], which is tailored to reduce the number of false positives. Other model checkers or bounded verifiers for LLVM IR are LLMC [2], RCMC [16], Serval [26], FauST [33] and SAW [8]. They can only check properties over a bounded state space, in contrast to our approach which uses deductive verification. PhASAR is a static analysis framework for LLVM IR [36]. Users specify arbitrary data-flow properties, and PhASAR then fully automatically tries to analyse these properties. The approach shows promising results, but as it is fully automatic, it also suffers from imprecisions that have to be manually filtered out. Lammich [18] formalises the semantics of LLVM IR, using it as the target language of the Refinement Framework in Isabelle. They do not analyse LLVM IR programs, but rather they derive correct by construction LLVM IR programs. Finally, verifying complex programs in the current VCLLVM/VerCors implementation heavily relies on pure functions. This is similar to approach of Paganoni and Furia [30] using predicates to verify Java bytecode.

7 Next Steps

As mentioned above, the current version of VCLLVM is still a prototype, and it needs to be extended with better support for more language features, control flow reconstruction, concurrency, and library inclusion.

Ultimately, the idea is not to use VCLLVM as a standalone tool to verify LLVM IR programs directly, but rather to use it as part of a larger infrastructure (called Pallas) that will provide deductive verification support for any programming language that can be compiled into LLVM IR. Figure 3 gives a visual representation of the Pallas infrastructure. It will define a generic specification format for contract specifications. For each source-level programming language supported by Pallas, a concrete contract specification syntax is defined to specify the desired program properties at the level of the source language, and then this should bee embedded into the generic contract specification format. The source to LLVM IR compiler is then used, combined with a compiler

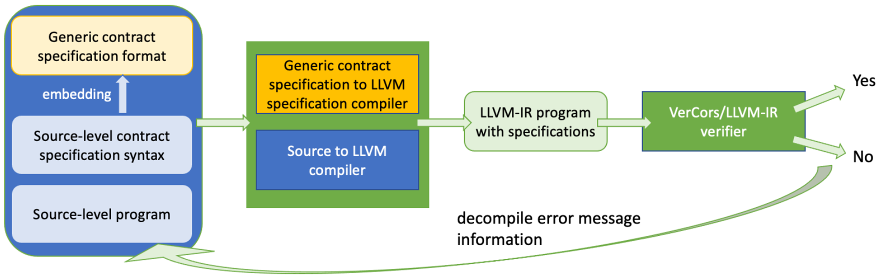
Generic contract specification format
embedding
Source-level contract specification syntax
Source-level program
Generic contract specification to LLVM specification compiler
Source to LLVM compiler
LLVM-IR program with specifications
VerCors/LLVM-IR verifier
Yes
No
decompile error message information

Fig. 3: Pallas Overall Idea

for the contract specifications in the generic contract specification format to the LLVM IR format. VCLLVM then enables VerCors to reason about the program. If verification succeeds, we know that the original source program satisfies the source-code-level contracts; if verification fails, the error message will be translated back into an error message for the source program.

Further research questions that we need to investigate to create the Pallas infrastructure are: (1) How to define a generic contract specification format that can capture program properties for a large class of source-level programming languages? (2) How to define a generic translation from the contract specification format into LLVM IR contract specifications, which can be parametrised by the compiler from a specific source language into LLVM IR? (3) How to provide effective feedback at the level of the source language if verification at the LLVM IR level fails by using decompilation techniques?

8 Conclusions

As a first step to solve the language-dependency problem of deductive verifiers, we propose to use the LLVM IR format as a generic format. This paper sketches the design of VCLLVM, a prototype implementation that enables deductive verification of LLVM IR programs, and we discuss the kind of examples that can already be verified. In future work, we will expand this into a deductive verification framework for any language that can be compiled into LLVM IR.

Data-Availability Statement

The artifact accompanying this paper can be found in [41].

References

[1] W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich. *Deductive Software Verification – The KeY Book*. Vol. 10001. Lecture Notes in Computer Science. Springer International Publishing, 2016. ISBN: 9783319498126. DOI: 10.1007/978-3-319-49812-6.

[2] F. 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*. Springer. 2021, pp. 690–703.

[3] A. Blanchard, F. Loulergue, and N. Kosmatov. "Towards Full Proof Automation in Frama-C Using Auto-active Verification". In: *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*. Ed. by J. M. Badger and K. Y. Rozier. Vol. 11460. Lecture Notes in Computer Science. Springer, 2019, pp. 88–105. DOI: 10.1007/978-3-030-20652-9_6. URL: https://doi.org/10.1007/978-3-030-20652-9_6.

[4] S. Blom, S. Darabi, M. Huisman, and W. Oortwijn. "The VerCors Tool Set: Verification of Parallel and Concurrent Software". In: *integrated Formal Methods 2017*. Ed. by N. Polikarpova and S. Schneider. LNCS 10510. Springer, 2017, pp. 102 –110. DOI: 10.1007/978-3-319-66845-1_7.

[5] J. Boyland. "Checking Interference with Fractional Permissions". In: *SAS*. Vol. 2694. LNCS. Springer, 2003, pp. 55–72.

[6] C. Cadar, D. Dunbar, and D. R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by R. Draves and R. van Renesse. USENIX Association, 2008, pp. 209–224. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[7] D. R. Cok. "OpenJML: JML for Java 7 by Extending OpenJDK". In: *NASA Formal Methods*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 472–479. ISBN: 978-3-642-20398-5.

[8] R. Dockins, A. Foltzer, J. Hendrix, B. Huffman, D. McNamee, and A. 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*. Springer. 2016, pp. 56–72.

[9] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. "KIV: overview and VerifyThis competition". In: *STTT* 17.6 (2015), pp. 677–694. ISSN: 1433-2787. DOI: 10.1007/s10009-014-0308-3. URL: https://doi.org/10.1007/s10009-014-0308-3.

[10] S. d. Gouw, F. S. de Boer, R. Bubel, R. Hähnle, J. Rot, and D. Steinhöfel. "Verifying OpenJDK's Sort Method for Generic Collections". In: *J. Autom.*

Reason. 62.1 (2019), pp. 93–126. DOI: 10.1007/s10817-017-9426-4. URL: https://doi.org/10.1007/s10817-017-9426-4.

[11] R. Hähnle and M. Huisman. "Deductive Software Verification: From Pen-and-Paper Proofs to Industrial Tools". In: *Computing and Software Science - State of the Art and Perspectives*. Ed. by B. Steffen and G. J. Woeginger. Vol. 10000. Lecture Notes in Computer Science. Springer, 2019, pp. 345–373.

[12] M. S. Hecht and J. D. Ullman. "Flow graph reducibility". In: *Proceedings of the fourth annual ACM symposium on Theory of computing*. 1972, pp. 238–250.

[13] H. A. Hiep, O. Maathuis, J. Bian, F. S. de Boer, M. C. J. D. van Eekelen, and S. de Gouw. "Verifying OpenJDK's LinkedList using KeY". In: *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings, Part II*. Ed. by A. Biere and D. Parker. Vol. 12079. Lecture Notes in Computer Science. Springer, 2020, pp. 217–234. DOI: 10.1007/978-3-030-45237-7_13. URL: https://doi.org/10.1007/978-3-030-45237-7_13.

[14] B. Jacobs and F. Piessens. *The VeriFast program verifier*. Tech. rep. CW520. Katholieke Universiteit Leuven, 2008.

[15] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. "VeriFast: A powerful, sound, predictable, fast verifier for C and Java". In: *NASA Formal Methods Symposium*. Ed. by M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi. Springer. 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4.

[16] M. Kokologiannakis, O. Lahav, K. Sagonas, and V. Vafeiadis. "Effective stateless model checking for C/C++ concurrency". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–32.

[17] N. Kosmatov, D. Longuet, and R. Soulat. "Formal Verification of an Industrial Distributed Algorithm: An Experience Report". In: *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*. Ed. by T. Margaria and B. Steffen. Vol. 12476. Lecture Notes in Computer Science. Springer, 2020, pp. 525–542. DOI: 10.1007/978-3-030-61362-4_30. URL: https://doi.org/10.1007/978-3-030-61362-4_30.

[18] P. Lammich. "Generating Verified LLVM from Isabelle/HOL". In: *10th International Conference on Interactive Theorem Proving (ITP 2019)*. Ed. by J. Harrison, J. O'Leary, and A. Tolmach. Vol. 141. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 22:1–22:19. ISBN: 978-3-95977-122-1. DOI: 10.4230/LIPIcs.ITP.2019.22. URL: http://drops.dagstuhl.de/opus/volltexte/2019/11077.

[19] C. Lattner and V. Adve. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86. DOI: 10.5555/977395.977673.

[20] K. Leino. “Accessible Software Verification with Dafny”. In: *IEEE Software* 34.6 (2017), pp. 94–97. DOI: 10.1109/MS.2017.4121212.

[21] LLVM Project. *LLVM Developer Policy: IR Backwards Compatibility.* https://llvm.org/docs/DeveloperPolicy.html{#}ir-backwards-compatibility. [Accessed 05-Dec-2022]. Dec. 2022.

[22] LLVM Project. *LLVM Language Reference Manual.* https://releases.llvm.org/15.0.0/docs/LangRef.html. [Accessed 05-Dec-2022]. Sept. 2022.

[23] LLVM Project. *opt - LLVM optimizer.* https://releases.llvm.org/15.0.0/docs/CommandGuide/opt.html. [Accessed 05-Dec-2022]. Sept. 2022.

[24] N. P. Lopes, J. Lee, C. Hur, Z. Liu, and J. Regehr. “Alive2: bounded translation validation for LLVM”. In: *PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021.* Ed. by S. N. Freund and E. Yahav. ACM, 2021, pp. 65–79. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[25] P. Müller, M. Schwerhoff, and A. Summers. “Viper - A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation. VMCAI.* Ed. by B. Jobstmann and K. R. M. Leino. Springer Berlin Heidelberg, 2016. DOI: 10.1007/978-3-662-49122-5_2.

[26] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang. “Scaling symbolic evaluation for automated verification of systems code with Serval”. In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 2019, pp. 225–242.

[27] P. W. O’Hearn. “Resources, concurrency and local reasoning”. In: 375.1–3 (2007), pp. 271–307.

[28] D. van Oorschot. *VCLLVM: A Transformation Tool for LLVM IR programs to aid Deductive Verification.* 2023. URL: http://essay.utwente.nl/96536/.

[29] W. Oortwijn, M. Huisman, S. Joosten, and J. van de Pol. “Automated Verification of Parallel Nested DFS”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* Springer. 2020, pp. 247–265.

[30] M. Paganoni and C. A. Furia. “Verifying Functional Correctness Properties at the Level of Java Bytecode”. In: *International Symposium on Formal Methods.* Springer. 2023, pp. 343–363.

[31] D. J. Pearce, M. Utting, and L. Groves. “An Introduction to Software Verification with Whiley”. In: *Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, China, April 7-12, 2018,*

Tutorial Lectures. Ed. by J. P. Bowen, Z. Liu, and Z. Zhang. Vol. 11430. Lecture Notes in Computer Science. Springer, 2018, pp. 1–37. DOI: 10.1007/978-3-030-17601-3_1.

[32] Z. Rakamaric and M. 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.* Ed. by A. Biere and R. Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 106–113. DOI: 10.1007/978-3-319-08867-9_7. URL: https://doi.org/10.1007/978-3-319-08867-9_7.

[33] H. Riener and G. 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.* Springer. 2012, pp. 234–240.

[34] M. Safari and M. Huisman. "Formal Verification of Parallel Stream Compaction and Summed-Area Table Algorithms". In: *Theoretical Aspects of Computing – ICTAC 2020.* Ed. by V. K. I. Pun, V. Stolz, and A. Simao. Springer, 2020, pp. 181–199. DOI: 10.1007/978-3-030-64276-1_10.

[35] M. Safari, W. Oortwijn, S. Joosten, and M. Huisman. "Formal verification of parallel prefix sum". In: *NASA Formal Methods Symposium.* Ed. by R. Lee, S. Jha, A. Mavridou, and D. Giannakopoulou. Springer, 2020, pp. 170–186. DOI: 10.1007/978-3-030-55754-6_10.

[36] P. D. Schubert, B. Hermann, and E. Bodden. "PhASAR: An Inter-procedural Static Analysis Framework for C/C++". In: *Tools and Algorithms for the Construction and Analysis of Systems.* Ed. by T. Vojnar and L. Zhang. Cham: Springer International Publishing, 2019, pp. 393–410. ISBN: 978-3-030-17465-1.

[37] M. Sitaraman and B. W. Weide. "A Synopsis of Twenty Five Years of RESOLVE PhD Research Efforts: Software Development Effort Estimation Using Ensemble Techniques". In: *ACM SIGSOFT Softw. Eng. Notes* 43.3 (2018), p. 17. DOI: 10.1145/3229783.3229794.

[38] VerCors team. *The VerCors Verifier Tutorial.* URL: https://vercors.ewi.utwente.nl/wiki/.

[39] Y. Zakowski, C. Beck, I. Yoon, I. Zaichuk, V. Zaliva, and S. Zdancewic. "Modular, compositional, and executable formal semantics for LLVM IR". In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473572. URL: https://doi.org/10.1145/3473572.

[40] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. "Formal verification of SSA-based optimizations for LLVM". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013.* Ed. by H. Boehm and C. Flanagan. ACM, 2013, pp. 175–186. DOI: 10.1145/2491956.2462164. URL: https://doi.org/10.1145/2491956.2462164.

[41] Ö. Şakar, D. van Oorschot, and M. Huisman. *Artifact for paper (First Steps towards Deductive Verification of LLVM IR)*. en. 2024. DOI: `10.4121/9C8C079E-A941-4A66-89D8-3462BF30FF05.V1`.