The VerCors Verifier: A Progress Report

Lukas Armborst[1], Pieter Bos[1], Lars B. van den Haak[2], Marieke Huisman[1], Robert Rubbens[1](✉), Ömer Şakar[1], and Philip Tasche[1]

[1] Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{l.armborst,p.h.bos,m.huisman,r.b.rubbens,o.f.o.sakar, p.b.h.tasche}@utwente.nl

[2] Software Engineering Technology, Technical University of Eindhoven, Eindhoven, The Netherlands

l.b.v.d.haak@tue.nl

Abstract. This paper gives an overview of the most recent developments on the VerCors verifier. VerCors is a deductive verifier for concurrent software, written in multiple programming languages, where the specifications are written in terms of pre-/postcondition contracts using permission-based separation logic. In essence, VerCors is a program transformation tool: it translates an annotated program into input for the Viper framework, which is then used as verification back-end. The paper discusses the different programming languages and features for which VerCors provides verification support. It also discusses how the tool internally has been reorganised to become easily extendible, and to improve the connection and interaction with Viper. In addition, we also introduce two tools built on top of VerCors, which support correctness-preserving transformations of verified programs. Finally, we discuss how the VerCors verifier has been used on a range of realistic case studies.

1 Introduction

With the ever-growing digitalisation of our society, we depend more and more on the reliability of the underlying software. To provide guarantees about this reliability, we need tools that can do a formal analysis directly at the implementation level of the software. The VerCors verifier [12] contributes to this goal: it enables the verification of pre-/postcondition contract specifications for (concurrent) programs, written in a range of different programming languages.

Work on the VerCors verifier started in 2011 [2], focussing initially on the verification of concurrent Java programs, using permission-based separation logic. Over time, VerCors has expanded into a verification environment that supports reasoning about programs in a wide range of different programming languages. An important design goal of the VerCors verifier was to make a tool that

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

© The Author(s) 2024
A. Gurfinkel and V. Ganesh (Eds.): CAV 2024, LNCS 14682, pp. 3–18, 2024.
https://doi.org/10.1007/978-3-031-65630-9_1

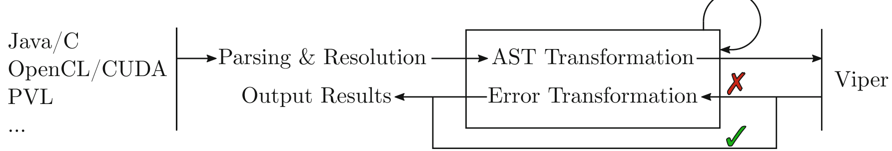

Fig. 1. Overview of the tool architecture. Tool interface boundaries are indicated with vertical lines. The circular arrow indicates that AST and error transformation steps might by applied multiple times. The red X (Color figure online) indicates a verification error, the green checkmark indicates succesful verification.

(1) would verify a program *as is*, i.e., without the need to manually simplify the implementation and only requiring additional verification annotations in comments, and (2) would have a high degree of automation, to make it accessible to a large group of potential users. Ultimately, VerCors should make verification available as a part of the build process, similar to type checking. VerCors is developed as a program transformation tool: it takes as input an annotated program, and it transforms this in multiple steps into input for the Viper framework [38], which is an intermediate representation framework for separation-logic-style specifications. An overview of the architecture is provided in Fig. 1. The transformation is set up in such a way that it is sound but incomplete: if Viper verifies the program, it is guaranteed that the original program satisfies its specification. However, if verification fails the program might or might not respect its verification annotations.

This paper reports on the recent steps that have been taken to further develop the VerCors verifier towards its ultimate goals. It describes in particular new developments on the VerCors verifier since 2017, when the last tool paper on VerCors was published [12]. Notable developments since then are:

- improved front-end support for programming languages such as Java, C and OpenCL, described in Sect. 2.1;
- added front-end support for other programming languages, such as Halide [44], SystemC [27], LLVM IR [31] and SYCL [58], described in Sects. 2.2 and 2.3;
- updated the internals of the tool to improve support for typing and transformation, as well as in the interaction with Viper, described in Sect. 3;
- a collection of transformation tools built on top of VerCors to step-wise derive verified, complex implementations, described in Sect. 4; and
- a wide range of practical case studies to understand how verification can be used in practice, described in Sect. 5.

2 New and Improved Language Support

This section describes the progress on programming languages supported by VerCors. First we describe new features that are provided for languages that were already supported by VerCors, namely Java and C/C++, as well as the improved support to reason about GPU kernels. Next, we describe new languages

for which direct support has been added to VerCors (JavaBIP and SYCL). The last subsection covers programming languages that are not directly supported by VerCors, but can be encoded – by VerCors itself, or by an external tool – into an existing VerCors language: SystemC, LLVM IR and Halide. For these encodings, we typically transform a program to PVL, which is VerCors' internal language. It is similar to Java, supporting classes and methods for example, but also has additional constructs such as parallel blocks, which we use to prototype new verification features.

2.1 Improved Existing Language Support

Java: Exceptions. As mentioned above, Java was the first programming language supported by VerCors. It has support for several non-trivial features of the language, such as the `import` statement, locks (specified using lock invariants), arrays, instance and `static` fields.

A missing feature that hindered practical applicability was the support to reason about exceptions. To improve this, we first added support for exceptional contracts using `signals` clauses. Similar to `ensures`, a `signals` clause specifies the postcondition that must hold when an exception of the indicated type is thrown. In addition, it can also specify properties over the object that is thrown.

Support for exception-related statements and modifiers such as `try_catch`, `throw` and `throws` is encoded by transforming them in several steps, to keep the implementation modular. For example, `throws` modifiers are encoded into `signals` clauses, and `try_catch` and throwing method calls are encoded into `goto`'s. After the transformations, the only primitives that remain are `goto`, `return`, `requires` and `ensures`. In addition, abrupt termination primitives such as `break` and `continue` are transformed into exceptional statements, such that they can be handled using the same code that encodes exceptional behaviour. For more details about the support for exceptions, see [47,48].

C/C++. Support for basic features of the C and C++ languages works similar to the verification of those features in Java. In particular, a C/C++ program can only be verified if it does not have undefined behaviour. However, also C-specific features had to be covered, such as *allocating* and *freeing* memory (`malloc` and `free`), *array initialisers*, *structures*, *casts* between primitive types and implicit type conversion rules. Furthermore, VerCors now uses the truncated [33] definition for division and modulo in the C, Java and C++ languages.

GPU Kernels: OpenCL/CUDA. VerCors initially supported verifying data race freedom and functional correctness of GPU kernels using barriers and atomic operations by manually encoding kernels into PVL (using parallel blocks) [13]. Support for verification directly at the level of the OpenCL [59] or CUDA [35] program has now been added, by implementing a translation from kernels into parallel blocks. In addition, support for both dynamic and static *local memory* (called *shared memory* in CUDA) is added, allowing verification of kernels that use faster data sharing for threads within the same workgroup. Support for local

and global memory fences for barriers is present for OpenCL, only allowing redistribution of memory permissions when the appropriate fence is used.

2.2 Newly Supported Frameworks

JavaBIP. VerCors has direct support for JavaBIP [11]. BIP [6] ("Behavior, Interaction, Priority") is a framework for rigorous system design. JavaBIP provides support for BIP as a Java library. Each JavaBIP class is modelled by a separate BIP state machine. This connection is made through annotations. The `@State` class annotation indicates the possible states, and the `@Transition` annotation indicates that a method makes a transition. Whenever a transition must be taken, the JavaBIP runtime engine looks up and executes the corresponding method. Essentially, the user declaratively specifies the state machine and implements the transition methods, and JavaBIP provides state machine behaviour.

The BIP methodology assumes that conditions on the behaviours of the system are encoded by the user in the BIP state machine, e.g. by adding guards to transitions, and by assuming implicit invariants in the state machine, such as "in state S, field f is positive". However, the JavaBIP platform does not provide tool support to check if an implementation actually guarantees these invariants. To address this shortcoming, we prototyped verification support for JavaBIP using VerCors [10]. In the JavaBIP state machine, the user makes implicit invariants explicit by adding contract annotations on the states and transitions. Guards and contracts are then verified deductively using VerCors, thus ensuring that the implementation corresponds to the assumptions for the BIP state machine.

SYCL. SYCL is a high-level programming language that enables the use of different heterogeneous devices in a single application [58]. It is built in C++ and targets different devices such as CPUs, GPUs and FPGAs. It abstracts away from the device-specific details (in contrast to e.g. OpenCL for GPUs), by building on top of existing (lower-level) APIs such as OpenCL, CUDA and HIP.

VerCors provides prototype verification support for a subset of SYCL, focussing on its basic and nd-range kernels, buffers and data accessors [60]. A contract is specified for the host function and SYCL kernel. VerCors automatically adds predefined specifications for the various SYCL data types and functions, and uses the kernel contract to automatically handle the permissions related to the data transfer and access through SYCL's buffer and accessor constructs.

2.3 Programming Languages Encoded into VerCors

SystemC. VerCors is able to verify embedded systems at the design stage, as it supports the hardware/software co-design language SystemC [27]. The VeSUV

tool [57] takes designs written in a widely-used subset of SystemC and encodes their semantics, as well as the scheduling semantics of SystemC, into PVL. The user can then add properties to the encoded PVL program and verify them normally with VerCors. This approach allows VerCors to verify both local and global safety properties and to reason about the timing behaviour of the system, which is typically difficult for deductive verifiers.

LLVM IR. VCLLVM is a prototype tool that adds support for LLVM IR [31] to VerCors [40,41]. Building verification support for LLVM IR is part of a larger project that aims to develop verification support for any programming language that compiles into LLVM IR. VCLLVM takes as input an annotated LLVM IR file. It uses the existing LLVM infrastructure to parse and analyse the program. The program and the annotations are then encoded by VCLLVM into input for VerCors, and VerCors is used for the verification.

Halide. HaliVer [24] is a tool that adds verification support for Halide [44] and uses VerCors as its verifier. Halide is a Domain Specific Language designed to write high-performance image and tensor processing code. Halide decouples the *algorithmic* part, which defines *what* should be computed, from the schedule, which defines how a computation should be optimised. HaliVer makes it possible to add and verify annotations that describe the behaviour of Halide programs. Verification can be done at two levels: (1) *front-end verification* encodes the algorithmic part of the Halide program directly into PVL, together with its annotations, to verify its functional correctness, while (2) *back-end verification* transforms the annotations to match the Halide-generated and optimised C code, which VerCors can then verify. This allows to verify complex optimised code, without formal verification of the whole Halide compiler. The HaliVer tool is integrated into the Halide compiler and transforms the annotations similar to how the compiler transforms the code.

3 VerCors Implementation Changes

In order to improve the user experience for VerCors users, as well as the extendability of the tool, some major updates to its implementation have been made. We describe the important changes.

Internal Transformation Steps. The effect of VerCors as a program transformer is achieved by a sequence of approximately eighty rewrite steps. Each step descends into the program tree recursively and rewrites nodes where appropriate. In earlier versions of VerCors there were several transformations containing over 1000 lines of code each, which made it hard to guarantee that they were correct rewrites. We reorganised the internal structure of VerCors and split those large transformations into multiple small rewrites. The smaller steps also help facilitate abstractions that newly supported languages like SYCL can build on.

Name Resolution. Earlier versions of VerCors used text names, which ended up with a large number of prefixes. If one was missing, it was hard to see which rewrite caused that. Schemes with de Bruijn-indexed names [17] are thwarted by declarations shifting around, and forgetting to account for it. Therefore, we made the rewriters blind to the declaration names. Instead, direct references to the referred declaration are stored in the tree. This means that names are resolved once at the start, and from then on there is nothing to resolve anymore: the name is a pointer to the declaration itself. When rewriting such a reference we look up the successor of the referent in a map. The circularity of this approach is resolved by storing the reference as a lazily evaluated value.

Typing Coercions. To ensure that all (intermediate) program trees during parsing and rewriting are correct, it is imperative that the program is well-typed. This is arranged by having each node in the tree assert typing constraints on its sub-nodes using the internal typing rules of VerCors. Moreover, certain rewriters need to know what typing rules were applied to the current node for it to be allowed in its current position, for example, if we want to store the sequence {`null, null, null`} in a variable of type `seq<int[]>`. This is achieved by temporarily storing the typing rule(s) that are applied to a node in the program tree, as a *coercion*. In this case, the sequence stores a coercion capturing that "`seq<null_type>` can be mapped to `seq<int[]>`, because `null_type` can be coerced to `int[]`". As a result, the rewriter for arrays only needs to consider places with the appropriate coercions.

Triggers. A known challenge in verification are quantifiers, which need instantiation in the proof. In the SMT community, triggers are used to manually provide hints about potential instantiations [21]. Initially, VerCors automatically generated triggers for quantifiers. However, for complicated examples it is important to have explicit control over triggers, to avoid matching loops [8]. Therefore, while VerCors still generates triggers, VerCors now also allows the user to specify triggers explicitly.

To enable the use of triggers for parallel block specifications, additional rewrites may be necessary. For a parallel block, the annotations are given per thread, and during the verification process these annotations are quantified over the range of all threads. However, in some cases this results in quantified formulas containing arithmetic expressions, which are not allowed in triggers. For example in the case of a flattened multi-dimensional array, we obtain specifications like: $\forall\, \mathsf{int}\ \mathsf{i}, \mathsf{int}\ \mathsf{j}; 0 \leq \mathsf{i} < 8 \wedge 0 \leq \mathsf{j} < 10 \wedge \mathsf{j}\%2 = 0 \Rightarrow \mathsf{A}[(\mathsf{j} * 8) + \mathsf{i}] > 0$. We would like to use the following trigger: $\mathsf{A}[(\mathsf{j} * 8) + \mathsf{i}]$. However, as arithmetic operators are not allowed in triggers, this trigger cannot be used. To fix this, VerCors can now automatically rewrite this expression to $\forall\, \mathsf{int}\ \mathsf{k}; 0 \leq \mathsf{k} < 8 * 10 \wedge (\mathsf{k}/8)\%2 = 0 \Rightarrow \mathsf{A}[\mathsf{k}] > 0$. This quantifier now has the following valid trigger: $\mathsf{A}[\mathsf{k}]$. This rewrite is general, and applies for most surjective mappings from variables to values.

Error Reporting. Errors that are reported about input programs are now modelled close to the input language. Earlier the tool reported simply that a formula

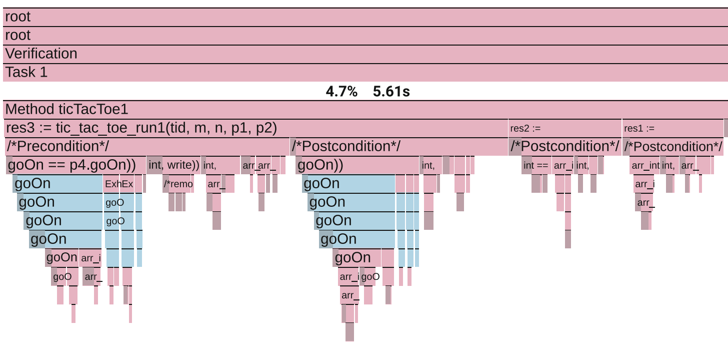

Fig. 2. Flame graph render of a verification profile

is false, or another technical error from the back-end. We extended the rewriters to indicate how errors reported in Viper [38] should be translated backwards in correspondence to the changes that occur in that rewriter. In certain cases such translations can consist of several steps, which have to be merged/combined, as rewriters build on abstractions within the internal VerCors language. Essentially, whenever a transformation creates an AST node that might cause a failure in the output AST, the transformation also has to define how to map the error back onto the input AST. The end result is that errors at the SMT level can be translated back to the input source level at the correct location.

Progress and Profiling. While VerCors is verifying a file, it now occasionally updates the user interface to show the proof goal it is working on. Since verification often gets stuck on a specific proof goal, this is helpful in diagnosing where the program needs further specifications or fixes. Currently this is reported in a rather technical manner, but we plan to soon adopt a better model, reporting in terms of the input program. This is inspired by the approach in the WP interface of Frama-C [7], where proof goals and their status are reported in line with the input file before the file is verified.

To keep the verify-edit-verify loop manageable, it also helps to be able to diagnose the verification time as a whole. For this purpose VerCors can now output a fine-grained profile, which contains timing information that can later be rendered to e.g. a flame graph as in Fig. 2. The tasks in the profile can be viewed as a tree structure, where a task is nested under its parent task. Tasks are divided up from global phases, down to the branch conditions under which a proof goal is verified. The detailed information about proof goals is supported through the symbolic execution back-end of Viper.

4 Deriving Verified, Optimised Programs

Program verification is a hard and challenging problem, and verifying a program that has been optimised for performance can be even harder. To alleviate this

problem, we have developed two program correctness-preserving optimisation tools on top of VerCors: Alpinist and VeyMont.

Alpinist. Alpinist is an annotation-aware GPU program optimiser [52]. Part of the GPU program development cycle is to incrementally optimise the GPU program for performance. These incremental optimisations are performed on the source level, prior to compilation. Such optimisations can introduce errors. To address this problem, Alpinist automatically applies frequently-used GPU program optimisations, notably loop unrolling, tiling, kernel fusion, iteration merging, matrix linearisation and data prefetching, in an *annotation-aware* manner, which means that besides transforming the GPU program itself, it also transforms the annotations. The provability of the resulting annotated optimised GPU program is preserved by this transformation. Alpinist works in four different phases: *parsing*, *applicability checking*, *transformation* and *output*. The strength of Alpinist's approach lies in particular in the *applicability checking*, where different analysis techniques, including deductive verification can be used to check whether the optimisation is indeed applicable, before applying it. An example of an applicability check is whether a loop can be safely unrolled a certain number of times (as specified by the user): Alpinist unrolls a loop n times only if it can prove that the loop is executed at least$n$ times.

VeyMont. VeyMont supports the derivation of correct parallel programs from sequential programs [16]. First, a sequential global program is verified. The sequential program has a restricted form, similar to choreographic programs and session types [26,36]. A sequential VeyMont program contains endpoints, communication statements between these endpoints, conditional statements, and loops, where for conditional statements all endpoints must agree on which branch is taken ("branch unanimity"). There are no local variables, instead all state is encapsulated by the endpoints. VeyMont can also generate permissions, however, this assumes a simple ownership structure without sharing.

After the global program is verified, VeyMont transforms it into a concurrent program, where an implementation is derived for each endpoint by projection [16]. For example, if an endpoint is in a receiving position in a communication statement, the projection will produce code that reads from the receiving end of the channel. If an endpoint is not involved in a communication, the projection will produce a no-op. The meta-theory behind VeyMont shows that this transformed program behaves in the same way as the sequential program [29]. In future work, we want to make VeyMont usable for a larger class of programs, in particular by providing support for the user to specify permissions, and by allowing parametrisation of global programs over the number of endpoints.

5 Case Studies

In order to evaluate and improve the usability and applicability of VerCors, we have developed a number of case studies using VerCors over the last years.

5.1 Tunnel Control Software Components

In collaboration with the company Technolution[1], several Java components of a tunnel control system were analysed with VerCors. The architecture of the software is governed by the Dutch tunnel standard specification (called BSTTI) [45]. First, we investigated the connection between the BSTTI and the implementation [42]. Next, we looked into a benign but unexplained runtime behaviour of the control software implementation [37]. Technolution suspected there was a concurrency bug in the code, but had not yet found a likely explanation.

After analysis and annotating the Java code, two possible explanations were found, and later confirmed by Technolution. First, there was a mutable internal data structure, which was accidentally aliased into a reference which was assumed to be immutable. Second, several methods allowed inspection and hence the leaking of an internal data structure, which was not designed to be thread safe. The collaboration with Technolution strengthened our ideas about what is needed for further adoption of verification in industry, as we not only encountered the problems ourselves, but also were able to confirm these findings with Technolution. These ideas are: language support has to be improved, code written without verification in mind is difficult to verify, and ultimately verification should be part of the development chain.

5.2 Verification of Red-Black Trees and their Parallel Merge

Another case study inspired by industrial code, this time from NLnet Labs[2], involved the verification of red-black (RB) trees. In the industrial C code, data was parsed by several threads concurrently, each constructing its own red-black tree. Afterwards, all those trees are merged in parallel into one. As a first step, Nguyen in his master thesis [39] implemented an RB tree in Java and verified parts of its functionality. This was later extended by verifying the delete functionality, as well as a version of the parallel merging process [4]. It uses a linked-list data structure to store batches of RB tree nodes, prepared by a producer thread and queueing for a consumer thread. This case study particularly highlights the use of two concepts supported by VerCors: (1) The producer-consumer pattern was proved using *ghost variables*, i.e. variables that only exist for specification and verification, and are not part of the executable code. While ghost variables are not unique to VerCors, the case study provides a useful example how they can assist in verifying concurrent programs in VerCors. (2) The delete operator was verified using the *separating implication* operator ("magic wand"), which is the separation logic counterpart to the logical implication "$\Rightarrow$" [14,53]. Many tools based on separation logic do not support the magic wand, but this case study shows its usefulness.

[1] https://www.technolution.com/.

[2] https://nlnetlabs.nl/.

5.3 GPU Case Studies

We developed several verification case studies for GPU code. Notably, we studied the verification of various prefix sum implementations, which is a frequently used library function for GPU kernels [51]. After that, we verified two other GPU algorithms (Parallel Stream Compaction and Summed-Area Table) that use prefix sum, to show how to verify code reusing existing verification results [50]. Initially, we verified encodings of the algorithms in PVL, but to show practical applicability of our approach, we also verified CUDA versions for most of them [49]. These case studies helped us to improve our GPU support and understand how these proofs work.

5.4 Student Projects

Several students have done case studies with VerCors. We find these student projects important, as they show the usability of VerCors for users who are not involved in the development of VerCors.

Sequential SCC Algorithm. The strongly connected components (SCC) algorithm finds the maximal subsets of nodes in a directed graph, such that every node in the component can reach any other node in the component, without leaving the component. It is an important ingredient for many model checking algorithms, and thus its correctness is essential. We had two student projects on the verification of a variation of Tarjan's SCC algorithm [56] in PVL. Hollander [25] provided an overall outline of the correctness proof, which was proven correct with VerCors, however using some unproven lemmas. Boerman [15] then followed up on this, and proved two of these remaining lemmas, to complete the soundness proof of the algorithm. In addition, Boerman identified several bottlenecks that slowed down the proof, and documented how they were resolved.

Distributed Locks. An implementation of a distributed re-entrant lock was verified to be memory safe and functionally correct by Ledelay [32]. The case study was provided by the company BetterBe[3]. To make verification tractable, Ledelay split up the implementation into four intermediate versions of increasing complexity, adding more aspects of the original code of BetterBe in each layer. The first layer was based on an earlier verified re-entrant lock [3]. The second layer added read/write functionality to the lock, and the third layer added an abstraction for a database. Finally, the fourth phase added a "fail-fast" optimisation, where a lock can safely skip a database query in certain cases. Layers one and two were fully verified. Verification of the later layers was not completed due

[3] https://www.betterbe.com/.

to time constraints. During verification, the multiple layers of abstraction sometimes made verification slower, or required additional verification annotations in other places. This was important input to improve the performance of VerCors.

Other Student Projects. Sessink verified (parts of) the implementation of the ArrayList class of Java's standard library [54]. This is relevant for verifying real-life code bases, which often make use of library features. Budde verified Kahn's topological sorting algorithm [18]. Like the SCC algorithm above, this is a base component in more complex procedures, such as task scheduling.

6 Conclusions, Related Work and Future Work

This paper gave an overview of recent work around the VerCors verifier. We described how we improved the support for programming languages that can be reasoned about, as well as the internals and verification support of the tool. We also discussed various case studies, which demonstrate the usability of the tool.

Related Work. There are several other deductive program verifiers for high-level programs, such as KeY [1], Dafny [34], OpenJML [20] Why3 [23], VeriFast [28], Frama-C [7], Whiley [43] RESOLVE [55] and the verifiers that are built on top of Viper, such as Nagini [22], Prusti [5] and Gobra [61]. The main characteristics that distinguish VerCors from these other tools are its focus on concurrency (only a few other verifiers, such as VeriFast and the Viper-based Gobra and Nagini, also support this), and its focus on extendability and support for many different programming languages and concurrency paradigms. However, we are often inspired by verification features and how they are built into other tools. There are also some tools that focus specifically on the analysis of GPU programs, such as GPUVerify [9] and Faisal [19]. They tailor their verification support specifically to GPU programs, whereas VerCors is fully general.

There also is related work on developing verification theories for concurrent software, such as Iris [30] and TaDa [46]. These form an inspiration for the verification logic supported by VerCors. However, our approach ultimately focuses on the applicability of our techniques, rather than covering all edge cases by developing a fully generic verification technique.

Future Work. Annotation generation is an important aspect of future work. HaliVer, Alpinist and VeyMont already address this for specific cases, but we also plan to develop techniques to generate annotations from scratch. Further, we would like to exploit the generality of VerCors further, to make it easier to support new programming languages. One future project is to investigate if we can use support for LLVM IR to develop verifiers for any language that compiles into LLVM IR. Finally, we continuously work on improving VerCors' usability.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: In: Deductive Software Verification – The KeY Book. LNCS, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6. ISBN: 9783319498126
2. Amighi, A., Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: The VerCors project: setting up basecamp. In: Programming Languages meets Program Verification (PLPV 2012), pp. 71–82. ACM (2012). https://doi.org/10.1145/2103776.2103785
3. Amighi, A.: Specification and verification of synchronisation classes in Java: a practical approach. Ph.D. thesis. University of Twente (2018). https://doi.org/10.3990/1.9789036544399
4. Armborst, L., Huisman, M.: Permission-based verification of red-black trees and their merging. In: 2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE), pp. 111–123 (2021). https://doi.org/10.1109/FormaliSE52586.2021.00017
5. Astrauskas, V., et al.: The Prusti project: formal verification for Rust. In: Deshmukh, J.V., Havelund, K., Perez, I. (eds.). NASA Formal Methods, pp. 88–108. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-06773-0_5. ISBN: 978-3-031-06773-0
6. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), pp. 3–12 (2006). https://doi.org/10.1109/SEFM.2006.27
7. Baudin, P., et al.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569
8. Becker, Nils, Müller, Peter, Summers, Alexander J..: The axiom profiler: understanding and debugging SMT quantifier instantiations. In: Vojnar, Tomáš, Zhang, Lijun (eds.) TACAS 2019. LNCS, vol. 11427, pp. 99–116. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17462-0_6
9. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2012), pp. 113–132. ACM (2012). https://doi.org/10.1145/2384616.2384625
10. Bliudze, S., van den Bos, P., Huisman, M., Rubbens, R., Safina, L.: Java-BIP meets VerCors: towards the safety of concurrent software systems in Java. In: Lambers, L., Uchitel, S. (eds.). Fundamental Approaches to Software Engineering, pp. 143–150. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30826-0_8. ISBN: 978-3-031-30826-0
11. Bliudze, S., Mavridou, A., Szymanek, R., Zolotukhina, A.: Exogenous coordination of concurrent software components with JavaBIP. In: Software: Practice and Experience, vol. 47, no. 11, pp. 1801–1836 (2017). https://doi.org/10.1002/spe.2495. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2495
12. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) Integrated Formal Methods 2017. LNCS, vol. 10510, pp. 102 –110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
13. Blom, S., Huisman, M., Mihelčić, M.: Specification and verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014). https://doi.org/10.1016/j.scico.2014.03.013. ISSN: 0167-6423

14. Blom, S., Huisman, M.: Witnessing the elimination of magic wands. Int. J. Softw. Tools Technol. Transfer **17**(6), 757–781 (2015). https://doi.org/10.1007/s10009-015-0372-3. ISSN: 1433-2787
15. Boerman, J.: Formal verification of a sequential SCC algorithm. MA thesis. University of Twente (2023). http://essay.utwente.nl/94474/
16. van den Bos, P., Jongmans, S.: VeyMont: parallelising verified programs instead of verifying parallel programs. In: Chechik, M., Katoen, J., Leucker, M. (eds.) Formal Methods, pp. 321–339. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-27481-7_19. ISBN: 978-3-031-27481-7
17. de Bruijn, N.G.: Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: Indagationes Mathematicae (Proceedings), vol. 75, no. 5, pp. 381–392. Elsevier (1972). https://doi.org/10.1016/1385-7258(72)90034-0
18. Budde, N.: Verified version of Kahn's topological sorting algorithm (2023). https://github.com/utwente-fmt/vercors/tree/5e3eb17/examples/concepts/algo/KahnsTopologicalSort.pvl. Accessed 17 Jan 2024
19. Cogumbreiro, T., Lange, J., Rong, D.L.Z., Zicarelli, H.: Checking data-race freedom of GPU kernels, compositionally. In: Silva, A., Leino, K.R.M. (eds.) CAV 2021. LNCS, vol. 12759, pp. 403– 426. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-81685-8_19
20. Cok, D.: OpenJML: software verification for Java 7 using JML, Open-JDK, and Eclipse. In: Dubois, C., Giannakopoulou, D., Mery, D. (eds.) 1st Workshop on Formal Integrated Development Environment, (F-IDE). EPTCS. 2014, vol. 149, pp. 79–92 (2014). https://doi.org/10.4204/EPTCS.149.8
21. Dross, C., Conchon, S., Paskevich, A.: Reasoning with triggers. Research Report RR-7986. INRIA, p. 29 (2012). https://inria.hal.science/hal-00703207
22. Eilers, M., Müller, P.: Nagini: a static verifier for Python. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 596–603. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_33
23. Filliâtre, J.-C., Paskevich, A.: Why3—where programs met provers. In: Felleisen, M., Gardner, P. (eds.) ESOP. LNCS, vol. 7792, pp. 125–128. Springer, Cham (2013). https://doi.org/10.1007/978-3-642-37036-6_8
24. van den Haak, L., Wijs, A., Huisman, M., van den Brand, M.: HaliVer: deductive verification and scheduling languages join forces. In: TACAS 2024. LNCS. Springer, Cham (2024)
25. Hollander, J.: Verification of a model checking algorithm in VerCors. MA thesis. University of Twente (2021). http://essay.utwente.nl/88268/
26. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems - ESOP 1998. LNCS, vol. 1381, pp. 122–138. Springer, Heidelberg (1998). https://doi.org/10.1007/BFB0053567
27. IEEE Standards Association. IEEE Std. 1666–2011, Open SystemC Language Reference Manual. IEEE Press (2011). https://doi.org/10.1109/IEEESTD.2012.6134619
28. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: a powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R.: NASA Formal Methods Symposium, pp. 41–55. Springer, Cham (2011). https://doi.org/10.1007/978-3-642-20398-5_4
29. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies - computing preconditions in choreographic programming. In: Sergey, I. (ed.) Programming Languages and Systems - 31st European Symposium on Programming,

ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, 2–7 April 2022, Proceedings. LNCS, vol. 13240, pp. 520–547. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99336-8_19
30. Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., Dreyer, D.: Iris from the ground up: a modular foundation for higher-order concurrent separation logic. J. Funct. Program. **28** (2018). https://doi.org/10.1017/S0956796818000151
31. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004, pp. 75–86. IEEE (2004). https://doi.org/10.5555/977395.977673
32. Ledelay, J.: Verification of Distributed Locks: A Case Study. MA thesis. University of Twente (2023). http://essay.utwente.nl/95192/
33. Leijen, D.: Division and Modulus for Computer Scientists (2003). https://www.microsoft.com/en-us/research/publication/divisionand-modulus-for-computer-scientists/
34. Leino, K.: Accessible software verification with Dafny. IEEE Softw. **34**(6), 94–97 (2017). https://doi.org/10.1109/MS.2017.4121212
35. Lindholm, L., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: a unified graphics and computing architecture. IEEE Micro **28**(2), 39–55 (2008). https://doi.org/10.1109/MM.2008.31
36. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023). https://doi.org/10.1017/9781108981491
37. Monti, R.E., Rubbens, R., Huisman, M.: On deductive verification of an industrial concurrent software component with VerCors. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles. ISoLA 2022. LNCS, vol. 13701, pp. 517–534. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-19849-6_29. ISBN: 978-3-031-19849-6
38. Müller, P., Schwerhoff, M., Summers, A.: Viper - a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation. VMCAI. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_2
39. Nguyen, H.: Formal verification of a red-black tree data structure. MA thesis. University of Twente (2019). http://essay.utwente.nl/77569/
40. van Oorschot, D.: VCLLVM: A Transformation Tool for LLVM IR programs to aid Deductive Verification". MA thesis. University of Twente (2023). http://essay.utwente.nl/96536/
41. van Oorschot, D., Huisman, M., Şakar, Ö.: First steps towards deductive verification of LLVM IR. In: FASE 2024, LNCS. Springer, Cham (2024)
42. Oortwijn, W., Huisman, M.: Formal verification of an industrial safety-critical traffic tunnel control system. In: Ahrendt, W., Tarifa, S.L.T. (eds.) Integreated Formal Methods (iFM) 2019. LNCS, vol. 11918. Springer, Heidelberg (2019). https://doi.org/10.1007/978-3-030-34968-4_23
43. Pearce, D.J., Utting, M., Groves, L.: An introduction to software verification with Whiley. In: Bowen, J.P., Liu, Z., Zhang, Z. (eds.) Engineering Trustworthy Software Systems - 4th International School, SETSS 2018, Chongqing, 7–12 April 2018, Tutorial Lectures. LNCS, vol. 11430, pp. 1–37. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-030-17601-3_1
44. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: a language and compiler for optimizing parallelism, locality, and recompu-

tation in image processing pipelines. In: ACM Sigplan Notices. PLDI 2013, vol. 48, no. 6, pp. 519–530 (2013). https://doi.org/10.1145/2491956.2462176

45. Rijkswaterstaat. Landelijke Tunnelstandaard (National Tunnel Standard). https://standaarden.rws.nl/link/standaard/6080. Accessed 17 Jan 2024
46. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: a logic for time and data abstraction. In: European Conference on Object-Oriented Programming (ECOOP). LNCS, vol. 8586. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44202-9_9
47. Rubbens, R.: Improving Support for Java Exceptions and Inheritance in VerCors. MA thesis. University of Twente (2020). http://essay.utwente.nl/81338/
48. Rubbens, R., Lathouwers, S., Huisman, M.: Modular transformation of Java exceptions modulo errors. In: Lluch-Lafuente, A., Mavridou, A. (eds.) Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, 24–26 August 2021, Proceedings. LNCS, Vol. 12863, pp. 67–84. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-85248-1_5
49. Safari, M., Huisman, M.: Formal verification of parallel prefix sum and stream compaction algorithms in CUDA. Theor. Comput. Sci. **912**, 81–98 (2022). https://doi.org/10.1016/J.TCS.2022.02.027
50. Safari, M., Huisman, M.: Formal verification of parallel stream compaction and summed-area table algorithms. In: Pun, V.K.I., Stolz, V., Simao, A. (eds.) Theoretical Aspects of Computing – ICTAC 2020, pp. 181–199. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-64276-1_10
51. Safari, M., Oortwijn, W., Joosten, S., Huisman, M.: Formal verification of parallel prefix sum. In: Lee, R., Jha, S., Mavridou, A., Giannakopoulou, D. (eds.) NASA Formal Methods Symposium, pp. 170–186. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-55754-6_10
52. Şakar, Ö., Safari, M., Huisman, M., Wijs, A.: Alpinist: an annotation-aware GPU program optimizer. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2022. LNCS, Vol. 13244, pp. 332–352. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-030-99527-0_18
53. Schwerhoff, M., Summers, A.J.: Lightweight support for magic wands in an automatic verifier. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, 5–10 July 2015, Prague. LIPIcs, vol. 37, pp. 614–638. Schloss Dagstuhl - Leibniz-Zentrum fur Informatik (2015). https://doi.org/10.4230/LIPICS.ECOOP.2015.614
54. Sessink, J.: Verified version of Java's ArrayList (2022). https://github.com/utwente-fmt/vercors/tree/5e3eb17/examples/concepts/arrays/ArrayList.java. Accessed 17 Jan 2024
55. Sitaraman, M., Weide, B.W.: A synopsis of twenty five years of RESOLVE PhD research efforts: software development effort estimation using ensemble techniques. ACM SIGSOFT Softw. Eng. Notes **43**(3), 17 (2018). https://doi.org/10.1145/3229783.3229794
56. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972). https://doi.org/10.1137/0201010
57. Tasche, P., Monti, R.E., Drerup, S.E., Blohm, P., Herber, P., Huisman, M.: Deductive verification of parameterized embedded systems modeled in SystemC. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.)25th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2024). LNCS, vol. 14500. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-50521-8_9

58. The Khronos SYCLWorking Group. SYCLTM 2020 Specification (revision 8). Specification. The Khronos Group (2023). https://registry.khronos.org/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf
59. The OpenCL 1.2 Specification. Khronos Group (2011)
60. Wittingen, E.: Deductive verification for SYCL. MA thesis. University of Twente (2023). https://purl.utwente.nl/essays/97976
61. Wolf, F.A., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J.C., Muller, P.: Gobra: modular specification and verification of Go programs. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. LNCS, vol. 12759, pp. 367–379. Springer, Heidelberg (2021). https://doi.org/10.1007/978-3-030-81685-8_17. ISBN: 978-3-030-81685-8

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.