The VerCors Tool Set: Verification of Parallel and Concurrent Software

Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn(✉)

University of Twente, Enschede, The Netherlands
{s.c.c.blom,s.darabi,m.huisman,w.h.m.oortwijn}@utwente.nl

Abstract. This paper reports on the VerCors tool set for verifying parallel and concurrent software. Its main characteristics are *(i)* that it can verify programs under different concurrency models, written in high-level programming languages, such as for example in Java, OpenCL and OpenMP; and *(ii)* that it can reason not only about race freedom and memory safety, but also about functional correctness. VerCors builds on top of existing verification technology, notably the Viper framework, by transforming the verification problem of programs written in a high-level programming language into a verification problem in the intermediate language of Viper. This paper presents three examples that illustrate how VerCors support verifying functional correctness of three different concurrency features: heterogeneous concurrency, kernels using barriers and atomic operations, and compiler directives for parallelisation.

1 Introduction

In a parallel or concurrent program, multiple program threads proceed in parallel while they access and write to a globally shared memory. Such programs are notoriously error-prone, because the set of possible program behaviours is exponential in the programs' size, containing all possible interleavings of the atomic steps of the individual threads. As a consequence, for developers it is easy to overlook a problem that occurs in only a few of these behaviours. Moreover, systematically testing all possible program behaviours is unfeasible for most concurrent programs. Nonetheless, parallel and concurrent programming is nowadays ubiquitous due to increased performance demands as well as the vast increase in availability of multi-core hardware. Tools and techniques are therefore needed that support the developers of such software to increase its reliability.

This paper discusses recent developments of the VerCors tool set, which aims to support developers in writing reliable concurrent software. VerCors allows *practical mechanised verification under different concurrency models*; notably *heterogeneous* concurrency (e.g. Java programs) and *homogeneous* concurrency (e.g. GPU kernels). Multiple widely-used languages with parallelism and concurrency features are targeted, such as Java, OpenCL, and OpenMP for C. It allows reasoning about data race freedom, memory safety, and functional properties of (possibly non-terminating) concurrent programs. Moreover, it can handle advanced language features such as compiler directives and atomic operations.

N. Polikarpova and S. Schneider (Eds.): IFM 2017, LNCS 10510, pp. 102–110, 2017.
DOI: 10.1007/978-3-319-66845-1_7

An earlier paper on the VerCors tool set has appeared in Formal Methods 2014 [5], where we showed how VerCors is used to prove data race freedom and basic functional correctness of concurrent Java [2] and OpenCL [6] programs. This paper extends on [5] and illustrates more advanced verification features of VerCors. First, we demonstrate our model-based approach to functional verification of concurrent Java programs, where an abstract model captures all concurrent behaviours of a program w.r.t. a set of shared variables [7,16]. We then use program logic-based verification to show the correspondence between the program and its abstraction, while algorithmic verification is used to reason about the abstract model. We also illustrate how VerCors is used to verify OpenCL kernels (OpenCL programs that run on GPUs) that use barriers and atomics for synchronisation [1]. Finally, programs with homogeneous threading are often constructed by developing a sequential program and adding suitable compiler directives, as is done in OpenMP. VerCors provides support to prove correctness of such compiler directives, i.e. ensuring that they will not change the functional behaviour of a program [4,10]. We also illustrate this by an example.

The VerCors tool set supports static verification in a design-by-contract fashion: programmers annotate their code and VerCors transforms verification of this annotated program into a verification problem in the intermediate verification language Silver [14]. The Viper verification technology (that works on Silver programs) is then used to verify the Silver specification with respect to its implementation. If this succeeds, we can conclude that the original program satisfies its annotations. Thus, the focus of VerCors is not so much on developing new verification technology, but rather on making existing verification technology usable for realistic programming languages and advanced language features. The specification language builds on *permission-based separation logic (PBSL)* [2,8], an extension of Hoare logic that explicitly considers where an object is stored in memory, which enables thread-modular verification of concurrent programs.

Section 2 provides a quick description of the tool architecture, focusing on its extendability. Section 3 discusses several examples to illustrate advanced features supported by VerCors. Section 4 concludes with a discussion of related and future work, and gives information about how to try VerCors yourself.

2 The VerCors Architecture

Our main goal is to make existing program verification technology usable for high-level programming languages and advanced language features. This is reflected in the design of VerCors, which is implemented as a collection of compiler transformations and uses the existing Viper technology as back-end [14], see Fig. 1. Viper supports the intermediate verification language Silver, which allows reasoning about programs with persistent mutable state, annotated with separation logic-style specifications. The compiler transformations are used to transform different high-level language/concurrency features into Silver code. The Viper technology provides two styles of reasoning: verification condition generation (via Boogie), and symbolic execution. The symbolic execution engine

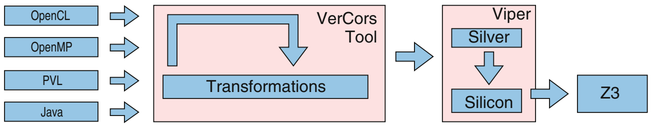

Fig. 1. The architecture of the VerCors tool set.

is the most powerful and provides support for e.g. quantified permissions, which we heavily rely upon. In earlier versions of VerCors, Chalice [13] was used as the main back-end, but its functionality is subsumed by Viper.

VerCors takes as input a program in a high-level programming language, annotated with JML-style specifications, and transforms this into verification problems encoded in Silver. The current input languages are Java, PVL, OpenCL, and OpenMP for C; it supports reasoning about the main concurrency-related features of these languages. The support for OpenCL covers only the verification of kernels, including barrier synchronisation and atomic operations, but not host code (which would mostly require engineering). PVL is a Java-like procedural toy language used for quick prototyping of new verification features. Notably, it has support for kernels and hostcode. VerCors also supports a substantial subset of OpenMP, essentially characterising deterministic parallel programming. The annotation language of VerCors is the same across all supported languages.

VerCors can easily be extended with new parallel or concurrent pointer languages, by providing a parser that transforms input programs and their specifications into the *intermediate language* of VerCors. All further program transformations are defined over the intermediate language of VerCors, thereby automatically providing verification support for the features of the extended language.

3 Verification Highlights

This section discusses three verification examples to illustrate the most interesting features supported by VerCors. For clarity of presentation the example annotations are somewhat simplified; the full, verifiable programs are available at http://www.utwente.nl/vercors. Also a detailed list of case studies and verified example programs is available, together with statistics about performance and required amounts of specification code relative to program code.

Model-Based Verification. In the context of heterogeneous threading, verification of functional properties is a major challenge and requires suitable abstractions. Our model-based verification technique captures the behaviour of a shared memory concurrent program by means of a *process algebra term with data* [7,16]. All accesses to the relevant shared memory locations are abstracted by *actions*. The process algebra term specifies the legal sequences of actions that are allowed to occur, and the program logic is used to verify that the process algebra term is indeed a correct program abstraction. Functional properties about the program can then be verified by reasoning algorithmically on the process algebra term.

We illustrate this on the parallel GCD challenge from the VerifyThis 2015 program verification competition [11]. The standard sequential Euclidean algorithm is described as a function `gcd` which, given two positive integers a and b, $\texttt{gcd}(a, a) = a$, $\texttt{gcd}(a, b) = \texttt{gcd}(a - b, a)$ if $a > b$, and $\texttt{gcd}(a, b) = \texttt{gcd}(a, b - a)$ if $b > a$. The parallel version we consider uses two concurrent threads: one thread to repeatedly decrease a when $a > b$, and one thread to repeatedly decrease the value of b when $b > a$. This process continues until a and b converge to $\texttt{gcd}(a, b)$.

```
int x, y;

guard y > 0 ∧ x > y; effect x = old(x) − old(y); action decrX();
guard x > 0 ∧ y > x; effect y = old(y) − old(x); action decrY();
guard x = y; action done();

requires x > 0 ∧ y > 0;
ensures x = y ∧ y = gcd(old(x), old(y));
process pargcd() := tx() ∥ ty();

process tx() := decrX() · tx() + done();
process ty() := decrY() · ty() + done();
```

Fig. 2. The process algebraic description of the `gcd` algorithm.

To prove that the parallel algorithm computes $\texttt{gcd}(a, b)$ we first model `gcd` as a process algebra term, named pargcd, by using two actions, named decrX and decrY. The decrX action corresponds to the assignment $x := x - y$ in the program code, and decrY corresponds to $y := y - x$. Action behaviour is defined in terms of *guard* and *effect* clauses, which logically describe the (guarded, conditional) effects of an action on the shared memory. A third action done indicates termination of the process term. Figure 2 shows the abstract model.

We use existing process-algebraic reasoning techniques to analyse the process pargcd: by giving any two positive integers as input, their `gcd` has been found when the action done has been performed. This is currently done by translating the analysis into an SMT problem, by encoding it into Silver. Finally, we prove the connection between pargcd and the concrete program code, presented in Fig. 3. In future work we plan to analyse the processes via the mCRL2 toolset.

The `calcgcd` function creates a new *model* named m via the invocation on line 4. The model m is *split* along the parallel composition tx() ∥ ty() on line 5 to match the forking of the two program threads T_0 and T_1 (where the body of thread T_1 is omitted for brevity). The thread T_0 requires that part of the model that executes the process term tx(); the thread T_1 requires the term ty(). The connection between program execution and process execution is made via action

```
requires $x > 0 \wedge y > 0$;
ensures \result = gcd($a, b$);
int calcgcd(int $a$, int $b$) {
  model $m$ := pargcd() with [$x := a, y := b$];
  split $m$ into ($\frac{1}{2}$, tx()) and ($\frac{1}{2}$, ty());
  invariant $inv(m.x \xhookrightarrow{1}_{\mathsf{p}} v * m.y \xhookrightarrow{1}_{\mathsf{p}} w * v > 0 * w > 0)$ {
    requires Proc($m, \frac{1}{2}$, tx()); ensures Proc($m, \frac{1}{2}, \varepsilon$);
    par $T_0$() {
      bool $run$ := true;
      loop-invariant $run$ ? Proc($m, \frac{1}{2}$, tx()) : Proc($m, \frac{1}{2}, \varepsilon$);
      while ($run$) {
        atomic ($inv$) {
          if ($m.x > m.y$) action decrX() { $m.x := m.x - m.y$; };
          if ($m.x = m.y$) action done() { $run$ := false; };
    } } }
    requires Proc($m, \frac{1}{2}$, ty()); ensures Proc($m, \frac{1}{2}, \varepsilon$);
    and par $T_1$ { $\cdots$ }
  }
  merge ($m, \frac{1}{2}, \varepsilon$) and ($m, \frac{1}{2}, \varepsilon$); finish $m$;
  return $m.x$;
}
```

Fig. 3. The annotated implementation of the parallel GCD algorithm.

annotations in the code. To this end, the actions decrX, decrY, and done are linked to concrete statements in the language via **action** blocks. Correctness of the connection is shown by applying the rules of our extended separation logic.

GPU Kernels and Atomics. VerCors supports verifying race freedom and functional correctness of GPU kernels that use atomic operations and barriers [1,6]. In a GPU kernel, threads are organised in workgroups, which consist of multiple threads. Threads within a workgroup can synchronise by means of a barrier; threads in different workgroups can only synchronise using atomic operations.

The VerCors tool set supports a kernel-specific version of PBSL; kernels are specified with the permissions available for them, in addition to their functional behaviour contract. The available permissions are distributed over the different workgroups, which in turn are specified with a permission distribution for its threads. We verify that these permission distributions are correct, meaning that kernels and workgroups do not distribute more permissions than are available. When threads within a workgroup synchronise on a barrier, they may redistribute permissions and exchange knowledge about their thread-local state.

We illustrate this approach on a kernel that calculates the sum of the elements of an array. The PVL encoding of this kernel is shown in Fig. 4 (the clause **context** P abbreviates **requires** P; **ensures** P). This example shows how race

```
 1 invariant A ≠ null ∧ m > 0 ∧ n > 0;
 2 context Perm(result, write) ∗ (\forall int i; 0 ≤ i < m ∗ n; Perm(A[i], read);
 3 requires result = 0;
 4 int calculate-sum(int m, int n, int[m∗n] A) {
 5   invariant outer(Perm(result, write)) {
 6     par kernel(int gid ∈ [0, . . . , m))
 7     context (\forall int i; 0 ≤ i < n; Perm(A[gid∗n+i], read); {
 8       int[1] temp := new int[1] { 0 };
 9       invariant inner(\array(temp, 1) ∗ Perm(temp[0], write)) {
10         par workgroup(int tid ∈ [0, . . . , n))
11         requires Perm(A[gid∗n+tid], read);
12         ensures tid = 0 ⇒ (\forall int i; 0 ≤ i < n; Perm(A[gid∗n+i], read)); {
13           atomic(inner) { temp[0] := temp[0] + ar[gid∗n +tid]; }
14           barrier (workgroup) {
15             requires Perm(A[gid∗n+tid], read);
16             ensures tid = 0 ⇒ (\forall int i; 0 ≤ i < n; Perm(A[gid∗n+i], read)); }
17           if (tid = 0) {
18             int tmp; atomic(inner) { tmp := temp[0]; }
19             atomic(outer) { result := result + tmp; }
20   }  · · ·  }
```

Fig. 4. Summing up the elements of the input array A.

freedom of kernels with barriers and atomics is verified, the interested reader can see the functional specification in [1]. The program uses two nested parallel blocks: the outer **par**-block resembles kernel execution, and the inner **par**-block resembles workgroup execution. First, each workgroup atomically adds the values in its part of the input array A to a *local* memory buffer *temp*. After writing to *temp*, each thread enters a barrier. After leaving the barrier, the first thread of each workgroup adds the local sum (stored in *temp*) to the *global* result. Each parallel block has a contract, denoting the requirements and the contributions of the workgroups and threads, respectively. In particular, each workgroup requires permission to read its share of A and each thread in a workgroup requires read permission to one entry of A. In the barrier, the read permission of each thread is transferred to the first thread in the workgroup.

To make the algorithm correct, addition to the shared intermediate result must be performed atomically (on line 20). In PVL this is expressed by putting the addition in an atomic block. Reasoning about atomic operations is an adaptation of the classical verification technique for atomic operations [15,19]. The specification language supports kernel and group invariants, which capture the behaviour of the atomic operations accessing the shared locations.

Deterministic Parallelism. Parallel programs are commonly written by using compiler directives, like done in OpenMP [17]. Compiler directives indicate code that may be executed in parallel, so that the compiler can generate parallelised

```
given seq⟨seq⟨int⟩⟩ data;
invariant m > 0 ∧ n > 0 ∧ p > 0 ∧ \matrix(M, m, n) ∧ \array(H, p);
context (\forall int i ∈ [0..m), j ∈ [0..n); Perm(M[i][j], read));
context (\forall int i ∈ [0..m), j ∈ [0..n); M[i][j] = data[i][j] ∧ 0 ≤ M[i][j] < p);
context (\forall int i ∈ [0..p); Perm(H[i], write));
ensures (\forall int k ∈ [0..p); H[k] = (\count int i ∈ [0..m), j ∈ [0..n); data[i][j] = k));
void histogram(int m, int n, int[m][n] M, int p, int[p] H) {
  for (int k := 0; k < p; k++)  context Perm(H[k], write); ensures H[k] = 0;
  { H[k] := 0; }
  for (int i := 0; i < m; i++)
  requires (\forall int k ∈ [0..p); Reducible(H[k], +));
  context Perm(M[i][j], read) ∗ 0 ≤ M[i][j] < p ∗ M[i][j] = data[i][j];
  ensures (\forall int k ∈ [0..p); Contribution(H[k], data[i][j] = k ? 1 : 0)); {
    for (int j := 0; j < n; j++) { H[M[i][j]] += 1; } } }
```

Fig. 5. The implementation of the histogram example, written in C.

code. VerCors provides support to prove that these compiler directives do not change the meaning of the program, meaning that functional correctness of the original program implies functional correctness of the parallelised program.

We illustrate this by means of a `histogram` example, see Fig. 5, which outputs an array H such that $H[k]$ contains the number of occurrences of the integer k in the input matrix M. We use VerCors to show that the **for**-loops can be parallelised without changing the functional program behaviour. We do this by specifying an *iteration contract* [4], which denotes the pre- and postcondition for each iteration of the loop. The iteration contract of the first loop expresses that each iteration k requires writing permission for $H[k]$ and sets $H[k]$ to zero. From the iteration contract we can derive that each loop iteration is independent, and thus that the loop can be parallelised without changing its functional behaviour. In a similar way, also for the second loop the iteration contract is used to capture independence of the iterations. The specification language provides extra annotations to deal with several typical scenarios; in this case, the Reducible and Contributes predicates are used to denote the reduction pattern.

4 Conclusion and Related Work

This paper gives a concise overview of the most interesting features of the VerCors toolset for verifying concurrent software. For more verification examples, statistical information, an indication of supported features, and for trying out the verification technology yourself, we refer to http://utwente.nl/vercors.

The VerCors tool set is currently used for teaching, as part of an advanced Master-level course on program verification. In addition, we also have several students working individually on interesting verification case studies, for example verifying the correctness of a parallel prefix sum implementation. Having non-developers of VerCors use the tool has been very useful to improve the maturity of the tool, to understand how people use the tool, and to see which features could be improved further. We are working on the development of a regression test suite, containing examples that *should* and that *should not* verify, which is automatically evaluated whenever the tool is updated. One particular challenge that we encountered is that we depend on the Viper framework, which is also still under development. Therefore, sometimes bug fixes for VerCors depend on Viper updates, and good communication with the group behind Viper is essential.

There exist several other tools for the verification of concurrent software, such as VeriFast [12] (for concurrent C and Java programs), VCC [9] (for C programs), Chalice [13] (for a concurrent toy language, not maintained anymore), Cave [18] (proving memory safety and linearizability), and GPUVerify [3] (automatic data race detection of GPU Kernels). The main distinguishing feature of the VerCors tool set is that it generalises the verification of concurrent software to a language-independent setting, where new front-ends can be added easily.

There are many directions we plan to explore to further increase usability of VerCors. We are currently investigating how our model-based verification technique can be used to reason about distributed software, focusing in particular on message passing. To improve scalability of the verification process we plan to experiment with different techniques for annotation generation and to generate meaningful error messages. Ultimately, our goal is to support complete programming languages, not just subsets. Since this is a large engineering effort, we hope to reuse existing verification technology as much as possible.

Acknowledgements. The work in this paper is partially supported by the ERC grant 258405 for the VerCors project, by the EU FP7 STREP 287767 project CARP, and by the NWO TOP 612.001.403 project VerDi.

References

1. Amighi, A., Darabi, S., Blom, S., Huisman, M.: Specification and verification of atomic operations in GPGPU programs. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 69–83. Springer, Cham (2015). doi:10.1007/978-3-319-22969-0_5
2. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1) (2015)
3. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: GPUVerify: a verifier for GPU kernels. In: OOPSLA, pp. 113–132. ACM (2012)
4. Blom, S., Darabi, S., Huisman, M.: Verification of loop parallelisations. In: Egyed, A., Schaefer, I. (eds.) FASE 2015. LNCS, vol. 9033, pp. 202–217. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46675-9_14
5. Blom, S., Huisman, M.: The VerCors Tool for verification of concurrent programs. In: Jones, C., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 127–131. Springer, Cham (2014). doi:10.1007/978-3-319-06410-9_9
6. Blom, S., Huisman, M., Mihelčić, M.: Specification and Verification of GPGPU programs. Sci. Comput. Program. **95**, 376–388 (2014)
7. Blom, S., Huisman, M., Zaharieva-Stojanovski, M.: History-based verification of functional behaviour of concurrent programs. In: Calinescu, R., Rumpe, B. (eds.) SEFM 2015. LNCS, vol. 9276, pp. 84–98. Springer, Cham (2015). doi:10.1007/978-3-319-22969-0_6

8. Bornat, R., Calcagno, C., O'Hearn, P.W., Parkinson, M.J.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
9. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03359-9_2
10. Darabi, S., Blom, S.C.C., Huisman, M.: A verification technique for deterministic parallel programs. In: Barrett, C., Davies, M., Kahsai, T. (eds.) NFM 2017. LNCS, vol. 10227, pp. 247–264. Springer, Cham (2017). doi:10.1007/978-3-319-57288-8_17
11. Huisman, M., Klebanov, V., Monahan, R., Tautschnig, M.: VerifyThis 2015: a program verification competition. Int. J. Softw. Tools Technol. Transfer (2016)
12. 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. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011). doi:10.1007/978-3-642-20398-5_4
13. Leino, K.R.M., Müller, P., Smans, J.: Verification of concurrent programs with chalice. In: Aldini, A., Barthe, G., Gorrieri, R. (eds.) FOSAD 2007-2009. LNCS, vol. 5705, pp. 195–222. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03829-7_7
14. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: a verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) VMCAI 2016. LNCS, vol. 9583, pp. 41–62. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49122-5_2
15. O'Hearn, P.W.: Resources, concurrency and local reasoning. Theoret. Comput. Sci. **375**(1–3), 271–307 (2007)
16. Oortwijn, W., Blom, S., Gurov, D., Huisman, M., Zaharieva-Stojanovski, M.: An abstraction technique for describing concurrent program behaviour. In: VSTTE (2017, to appear)
17. OpenMP Architecture Review Board, OpenMP API Specification for Parallel Programming. http://openmp.org/wp/. Accessed 18 Oct 2016
18. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14295-6_40
19. Vafeiadis, V.: Concurrent separation logic and operational semantics. In: MFPS. ENTCS, vol. 276, pp. 335–351 (2011)