The VerCors Tool for
Verification of Concurrent Programs

Stefan Blom and Marieke Huisman

Formal Methods and Tools, University of Twente, The Netherlands
{s.blom,m.huisman}@utwente.nl

Abstract. The VerCors tool implements thread-modular static verification of concurrent programs, annotated with functional properties and heap access permissions. The tool supports both generic multithreaded and vector-based programming models. In particular, it can verify multithreaded programs written in Java, specified with JML extended with separation logic. It can also verify parallelizable programs written in a toy language that supports the characteristic features of OpenCL. The tool verifies programs by first encoding the specified program into a much simpler programming language and then applying the Chalice verifier to the simplified program. In this paper we discuss both the implementation of the tool and the features of its specification language.

1 Introduction

Increasing performance demands, application complexity and explicit multi-core parallelism make concurrency omnipresent in software applications. However, due to the complex interferences between threads in an application, concurrent software is also notoriously hard to get correct. Therefore, formal techniques are needed to reason about the behavior of concurrent programs. Over the last years, program logics have proven themselves to be useful to reason about sequential programs. In particular, several powerful tools for JML have been developed [5]. These techniques now are mature enough to lift them to concurrent programs.

The VerCors tool supports the thread-modular verification of multithreaded programs. Modularity is achieved by specifying for each thread which variables on the heap it can access, by means of access permissions, which can be divided and combined, but not duplicated [8]. To read a location, any share of the access permission to that location suffices. To write a location a thread needs 100% of the access rights. Hence, if a thread has write permission to a location, no other thread can read that location simultaneously. Moreover, if a thread has read permission to a location, other threads can also only read this location. Thus specifications that are sufficiently protected by permissions are interference-free. Moreover, verified programs cannot contain data races.

Just as multi-core processors are ubiquitous, the same applies to GPU hardware. Therefore, the VerCors tool also provides the functionality to reason about kernels running on a GPU, where a large number of threads execute the same instructions, each on part of the data.

2 Design of the VerCors Tool

Rather than building yet another verifier, the VerCors tool leverages existing verifiers. That is, it is designed as a compiler that translates specified programs to a simpler language. These simplified programs are then verified by a third-party verifier. If there are errors then the error messages are converted to refer to the original input code.

Figure 1 shows the overall architecture of the tool. Its main input language is Java. For prototyping, we use the toy language PVL, which is a very simple object-oriented language that can express specified GPU kernels too. The C language family front-end is work-in-progress, but will support OpenCL in the near future. We mainly use Chalice [10], a verifier for an idealized concurrent programming language, as our back-end, but for sequential programs we also use the intermediate program verification language Boogie [1].

Fig. 1. VerCors tool architecture.

The implementation of the tool is highly modular. Everything is built around the Common Object Language data structure for Abstract Syntax Trees. For Java and C, parsing happens in two passes. In the first pass an existing ANTLR4 [13] grammar is used to convert the program into an AST while keeping all comments. In the second pass those comments that contain specifications are parsed using a separate grammar. This prevents us from having to maintain heavily modified grammars and makes it much easier to support multiple specification languages. The process of encoding programs consists of many simple passes. Obviously, this impacts performance, but it is good for reusability and checkability of the passes. Our back-end framework allows switching between different versions, by setting up their command line execution using environment modules, a system for dynamic access to multiple versions of software modules [11].
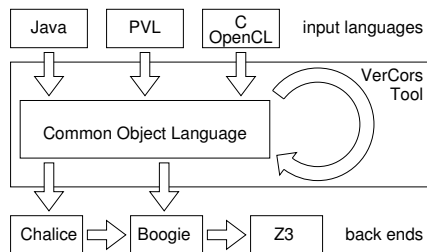
3 The VerCors Specification Language

The VerCors specification language has JML as a starting point, and adds features from Chalice, and from Hurlin's permission-based separation logic for concurrent Java [8], in order to be equally expressive as Hurlin's logic.

Using JML as a starting point allows to reuse existing JML specifications. However, JML's support for framing (i.e., modifies clauses) is not precise enough to be used in a concurrent setting. Instead we use access permissions $\mathsf{Perm}(e, \pi)$, where e is an expression denoting a location on the heap (a field in Java) and π is a percentage. To specify properties of the value stored at the location we just refer to the location in our formulas. Thus, we are forced to check that every expression is *self-framed*, i.e., we need to check that only locations for which we have access permission are accessed. This is different from classical separation

logic, which uses the PointsTo primitive, which has an additional argument that denotes the value of the location and cannot refer to the location otherwise. We prefer the Perm primitive because it fits JML and Chalice best. The VerCors tool supports PointsTo as syntactic sugar, which can be extended to full support. Moreover, it is proven that the two logics are equivalent [12]. Another feature of our logic is the notion of thread-local predicates, which are used to axiomatize the lockset predicate that keeps track of the locks held by the current thread [8].

Like Chalice, the VerCors tool disallows disjunction between resources. It does so by distinguishing the type resource from the type boolean. Thus, boolean formulas allow all logical operators and quantifications, while resource formulas are limited to the separating conjunction, separating implication (magic wand), and universal quantification. In method contracts, pre- and postconditions are of type resource.

VerCors' specification language uses several features that are not natively present in Chalice and thus have to be encoded. Resource predicates can have an arbitrary number of arguments, whereas Chalice only allows the implicit **this** argument. This is encoded by (partially) translating the formulas to witness objects. That is, instead of passing arguments to a predicate, we put the arguments in an object and define a predicate (without arguments) on that object. This translation also turns proof construction annotations into method calls. Magic wands are encoded using a similar strategy of defining witness objects [3]. By encoding complex specifications as data structures with simple specifications, we gain the ability to verify complex specifications with existing tools. However, these existing tools have no specific support for our data structures. Therefore, we also have to provide proof scripts to guide the proof search in the encoded program.

Below, we show a small example of a program in PVL that computes the fibonacci numbers by forking new threads instead of making recursive calls.

```
   class Fib { static int fib (int n)=n<2?1:fib(n−1)+fib(n−2);
2    int input, output;
     requires perm(input,50) ∗ perm(output,100);
4    ensures  perm(input,50) ∗ perm(output,100) ∗ output=fib(input);
     void run() { if (input<2) { output := 1; }
6              else { Fib f1 := new Fib; f1.input := input−1;
                     Fib f2 := new Fib; f2.input := input−2;
8                    fork f1; fork f2;
                     assert f1.input=input−1 ∗ f2.input=input−2;
10                   join f1; join f2;
                     output := f1.output + f2.output; }}}
```

Note that we use Chalice notation for fractions: 50 means read-only and 100 means write access. Also note how on line 9, we use an assert to remind the prover that because we can read the inputs to the threads, these inputs cannot change. The Java version of this example is much longer and can be found on the tool's website [14].

In addition to verification of Multiple Instruction Multiple Data programs, the VerCors tool also supports verification of Single Instruction Multiple Data

programs. Specifically, it supports reasoning about GPU kernels written in PVL. The concept of a kernel is that a large number of threads, divided over one or more working groups, all execute the same code, but each on part of the data. These computations cannot synchronize, except for barrier synchronization of the threads within a working group. Due to the lack of other synchronization primitives, the resources available for redistribution at a barrier are precisely those available to a working group at the start of the computation. This is reflected by the fact that the required resources upon entering a barrier are deduced by our tool instead of being specified by the user. Moreover, it means that in future versions we can simplify the permission model to three values: no access, read access, full access. Our kernel logic imposes proof obligations to ensure that all resources are always properly distributed [4]. The tool verifies these proof obligations by encoding them as specified methods and classes.

Below, we show a small example of a kernel. It displays a typical case: first each of the **gsize** threads computes a value based on an unknown function f and its identifier **tid**. Then the threads synchronize using a barrier and add their own result to that of the preceding thread to get their final result:

```
    global  int [gsize ] x,  y;
2   requires  perm(x[tid],100)  ∗ perm(y[tid],100);
    ensures   perm(x[tid],100)  ∗ (0<tid & tid<gsize −> x[tid]=f(tid)+f(tid−1));
4   void  main(){
       y[tid ]  := f(tid );
6     barrier (global){
          requires  y[tid]=f(tid );
8         ensures perm(x[tid],100)  ∗ perm(y[tid],50)  ∗ perm(y[(tid−1) mod gsize],50);
          ensures y[tid]=f(tid )  ∗ (tid>0 −> y[tid−1]=f(tid−1)); }
10     if  (tid>0) { x[tid ]  := y[tid]+y[tid−1]; } }
```

4   Conclusion

This paper gives a brief overview of the VerCors tool set and its specification language. The main application areas of the tool are MIMD programs written in Java, using Java's concurrency library, and SIMD applications, such as OpenCL kernels. The tool website [14] contains additional information such as our collection of verified examples, which can be tested with the online version of the tool. These examples demonstrate reasoning about the fork/join pattern, reentrant locks, and about magic wands in specifications. Additionally, there are also several verified kernel examples.

There are several other static verifiers that support reasoning about MIMD programs, such as VCC [6] for C, VeriFast [9] for C and Java, jStar [7] for Java, and Chalice [10] for an idealized concurrent language. The VCC tool has its own permission system and does not use separation logic. The VeriFast and jStar tools both use classical separation logic, with jStar being more limited (e.g. no support for fractional permissions). The Chalice tool, like VerCors uses implicit dynamic frames, which can be seen as a variant of separation logic [12]. The distinguishing feature of the VerCors tool compared to the ones above is that it

supports specifications using the magic wand operator. Moreover, VerCors has support for other concurrency models, such as the SIMD model used for GPU kernels. Memory safety for kernels can also be checked with GPUVerify [2], but additionally, VerCors can check functional correctness of kernels.

At the moment, the tool requires a considerable amount of annotations to verify a program. To reduce this, we will work on automatic generation of specifications and also on identifying and implementing useful default specifications and syntactic sugar. To turn the tool into a full-fledged verification tool, we have to add support for reasoning about *e.g.*, exceptions. Moreover, we will continue the work on the C parser, so the tool can verify OpenCL.

References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *LNCS*, pages 364 – 387. Springer, 2005.
2. A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA'12*, pages 113–132. ACM, 2012.
3. S. C. C. Blom and M. Huisman. Witnessing the elimination of magic wands. Technical Report TR-CTIT-13-22, Centre for Telematics and Information Technology, University of Twente, Enschede, November 2013.
4. S. C. C. Blom, M. Huisman, and M. Mihelcic. Specification and verification of gpgpu programs. Technical Report TR-CTIT-13-21, Centre for Telematics and Information Technology, University of Twente, Enschede, November 2013.
5. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
6. E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 23–42. Springer, 2009.
7. D. DiStefano and M. Parkinson. jStar: Towards practical verification for Java. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 213–226. ACM Press, 2008.
8. C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.
9. B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW520, Katholieke Universiteit Leuven, 2008.
10. K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*, pages 195–222. Springer, 2009.
11. The environment modules project. `http://modules.sourceforge.net`.
12. M. Parkinson and A. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3:01):1–54, 2012.
13. T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2013.
14. The vercors tool online. `http://www.utwente.nl/vercors/`.