Title: **CIVL: The Concurrency Intermediate Verification Language**

Authors: Stephen F. Siegel, Matthew B. Dwyer, Ganesh Gopalakrishnan, Ziqing Luo, Zvonimir Rakamarić, Rajeev Thakur, Manchun Zheng, Timothy K. Zirkel

Kind: Technical Report UD-CIS-2014/001

Status: *Submitted for publication*

Verified Software Laboratory
Department of Computer and Information Sciences
University of Delaware
Newark DE 19716
USA
`http://vsl.cis.udel.edu`

CIVL: The Concurrency Intermediate Verification Language

Stephen F. Siegel[1], Matthew B. Dwyer[2], Ganesh Gopalakrishnan[3], Ziqing Luo[1], Zvonimir Rakamarić[3], Rajeev Thakur[4], Manchun Zheng[1], and Timothy K. Zirkel[1]

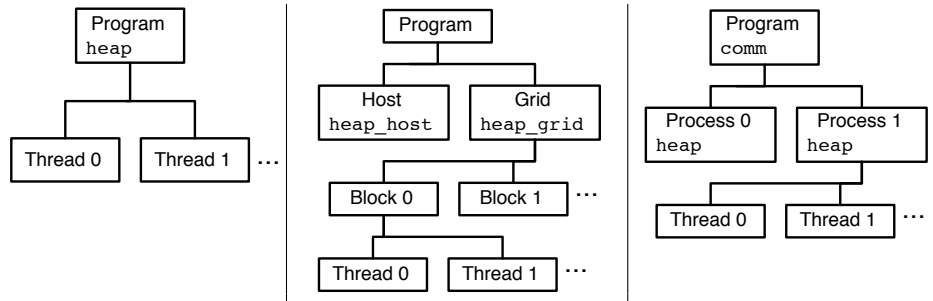
[1] Department of Computer and Information Sciences, University of Delaware, USA, {siegel|ziqing|zmanchun|zirkeltk}@udel.edu
[2] Department of Computer Science and Engineering, University of Nebraska - Lincoln, USA, dwyer@cse.unl.edu
[3] School of Computing, University of Utah, USA, {ganesh|zvonimir}@cs.utah.edu
[4] Mathematics and Computer Science Division, Argonne National Laboratory, USA, thakur@mcs.anl.gov

Abstract. Recent years have witnessed an explosion in the number of programming languages and language extensions dealing with concurrency. Examples include message-passing libraries (MPI-3), multithreading and GPU language extensions such as OpenMP, Pthreads, OpenCL, and CUDA, and entirely new languages such as Chapel. This multitude creates a serious challenge for developers of software verification tools: it takes enormous effort to develop such tools, but each development effort typically targets one small part of the concurrency landscape, with little sharing of techniques and code between efforts. To address this problem, we present CIVL: a Concurrency Intermediate Verification Language. CIVL provides a concurrency model sufficiently flexible to represent programs in a wide variety of parallel languages, including those listed above. We have realized CIVL as a dialect of C with new primitives for concurrency and others to facilitate specification and verification. We have also developed a tool that combines model checking and symbolic execution to verify or refute a number of properties of CIVL programs, such as absence of deadlock and assertion violations, and functional equivalence.

1 The CIVL Framework

The **CIVL framework** encompasses (1) the programming language **CIVL-C**, a dialect of C with additional primitives supporting concurrency, specification, and modeling; (2) verification and analysis tools, including a symbolic execution-based model checker for checking various properties of, or finding defects in, CIVL-C programs; and (3) tools that translate from many commonly used languages/APIs to CIVL-C.

Our goal is for CIVL-C to be an effective intermediate representation for verification. A C program using MPI [16], CUDA [11], OpenMP [7], OpenCL [21], or another API (or even some combination of APIs), will be automatically translated into CIVL-C and then verified. The advantages of such a framework are




```
$heap heap;
// shared vars
void Thread(int id) {
  // private vars
  ...
}
$proc threads[n];
for (i=0; i<n; i++)
  threads[i] =
    $spawn Thread(i);
for (i=0; i<n; i++)
  $wait threads[i];
```

(a) OpenMP

```
// shared state
void Host() {
 $heap heap_host;...
}
void Grid() {
 $heap heap_grid;
 // global vars
 void Block(int bid) {
  // shared vars
  void Thread (int tid) {
   // thread-local vars
}}} ...
```

(b) CUDA

```
$comm comm; // buffers
void Process(int pid) {
 $heap heap;
 // process variables
 void Thread(int tid) {
  // private vars
  ... $send, $recv ...
 }
 // spawn threads...
}
// spawn processes
// ...
```

(c) MPI/Pthreads

Fig. 1. Three concurrency models and their CIVL representations

clear: the developer of a new verification technique could implement it for CIVL-C and then immediately see its impact across a broad range of concurrent programs. Likewise, when a new concurrency API is introduced, one only needs to implement a translator from it to CIVL-C in order to reap the benefits of all the verification tools in the platform. Programmers would have a valuable verification and debugging tool, while API designers could use CIVL as a "sandbox" to investigate possible API modifications, additions, and interactions.

We currently have working versions of (1) and (2), while (3) remains for future work. For now, we are constructing CIVL-C programs by hand.

To illustrate the challenges involved, consider three widely used and evolving concurrency APIs: OpenMP, CUDA, and MPI. The models upon which these APIs are based are illustrated in Fig. 1, together with code skeletons showing how each can be represented in CIVL-C.

OpenMP is based on a shared memory model, including a single, shared heap. A program begins executing with a single thread. Pragmas indicate regions where work can be done in parallel; threads are spawned upon entering the region and joined back into a single thread at the end of the region. A *parallel for loop* is an example of such a region; the loop iteration space is partitioned among the threads in a way specified by the user or left up to the runtime system. The

programmer may also specify which variables are to be shared by the threads, and which are to be *private* (i.e., each thread will have its own copy).

CUDA and OpenCL are based on a complex hierarchical model (we follow the CUDA terminology in this paper). In this model, there is a *host* (CPU), and some number of *devices* (GPUs). Each device executes a *grid*, which is composed of *blocks*, which are composed of *threads*. There are parts of the state that are local to each part of this hierarchy; e.g., a thread can access the "shared memory" on its block, but not that of another block. There is a heap on the host and one for each device. Transferring data between host and device requires function calls, much like message passing.

MPI is based on an asynchronous message-passing model. An MPI program consists of concurrently executing *processes*, each with its own private memory, including its own heap. Typically, the processes exist for the life of the program, but they can also be created and destroyed dynamically. Communication and synchronization take place through explicit calls to functions in the MPI library. Moreover, each MPI process may be multithreaded (using, e.g., OpenMP or Pthreads); these so-called *hybrid* parallel programs are becoming increasingly common. Fig. 1(c) shows the structure of a hybrid MPI/Pthreads program.

While these three models appear quite different, there are some common themes. Each model has some notion of *process* (or thread); each has some hierarchical decomposition of the state into what we will call *scopes*. Both *processes* and *scopes* are dynamic concepts: they may be created and destroyed during an execution. CIVL-C incorporates these concepts as follows:

- like standard C, programs are lexically scoped and this static scope hierarchy can have arbitrary depth;
- unlike standard C, functions can be defined in any scope;
- the `$spawn` command (all CIVL-C primitives not in C begin with `$`) is like a function call, but spawns a new process executing the function and returns immediately a handle (of type `$proc`) to the new process;
- `$wait` takes an expression of type `$proc` and blocks until the referenced process has terminated; all references to the process are then replaced with the "undefined" value.

Formally, the state of a CIVL-C program consists of a set of *dynamic scopes* and a set of *process states*. Each dynamic scope δ is an *instance* of one static scope $\sigma = \sigma(\delta)$ and assigns a value to each variable declared in σ. The dynamic scopes are arranged in a tree so that the map that sends δ to $\sigma(\delta)$ is a tree homomorphism. Each process state consists of a *call stack*, which is a sequence of *frames*; each frame is an ordered pair consisting of a reference to a dynamic scope and a program location. New dynamic scopes are added whenever control enters a new static scope (including function calls and spawns); they are removed when they become unreachable. A process is added by a `$spawn` and removed when it has terminated and there is no reference to it in the state.

The language also provides certain abstract datatypes which are useful for modeling purposes. For example, there is a `$heap` datatype for representing a heap, together with functions `$malloc` and `$free`, which are similar to their C

counterparts, but take an extra argument, which is a pointer to a heap. This allows a program to have multiple, and scope-local, heaps.

The datatype `$comm` is an abstraction for a "communication universe" in a message-passing system. It is similar but more general than the *communicator* concept of MPI. It bundles together a set of *places* with any number of processes occupying each place, and a FIFO queue of buffered messages for each ordered pair of places. A communicator is initialized by specifying one process for each place, but processes can be added to places dynamically. Messages are sent from a place to a place, i.e., message meta-data includes a source and destination place but does not specify which process sent the message. A hybrid MPI program in which multiple threads on a process send and receive messages on behalf of that process maps naturally to this abstraction.

2 Tools

The current version of the CIVL platform, v0.7, provides command line tools to

1. *run* a CIVL-C program, using either random numbers or user interaction to resolve nondeterminism;
2. *verify* a CIVL-C program is free of deadlock, assertion violations, bounds violations, illegal pointer dereferences, divisions by 0, and improper uses of malloc/free, among other things; and
3. *replay* traces for counterexamples found during verification, which can show every execution step and the value of every variable at each state.

All of the tools use symbolic execution: the values assigned to variables are symbolic expressions and a path condition variable is used to keep track of decisions made at nondeterministic points. Inputs are initialized with unique symbolic constants. The use of symbolic execution also enables equivalence checking: functional equivalence of two versions of an algorithm can be specified by encoding each in its own function such that both functions read the same input; a main function then invokes both functions and asserts their outputs agree [29].

The verifier performs a depth-first search of the state space ("model checking"). It incorporates a number of CIVL-specific reduction techniques generalizing techniques (especially [13] for dealing with pointers) used in other model checkers; these can dramatically reduce the work required to determine whether the properties hold—in many cases to a single interleaving. CIVL states are immutable so they can easily share common subcomponents; this greatly enhances memory and time performance, and simplifies the process of saving states.

The tool set is implemented in Java 7. We have extended the ANTLR-based C front-end ABC [1] to preprocess, parse, analyze and construct an abstract syntax tree for CIVL-C. This AST is also fully accessible through an API, so other front-ends could also be used. The symbolic execution library SARL [28] is used to perform the symbolic manipulations and reasoning; SARL in turn uses the automated theorem prover CVC3 [6] and clj-ds (persistent data structures from Clojure) [9]. The platform is freely available under the GNU Public License from

`http://vsl.cis.udel.edu/civl`. Complete, pre-compiled packages (including the dependencies) are distributed for OS X and Linux systems. The packages also include a user's manual and over 100 examples and tests.

Limitations. Currently, all of C's integer types other than `_Bool` and `char` are mapped to a CIVL type which corresponds to the mathematical integers. Similarly, C's floating types are mapped to the mathematical reals. The bit-wise operations in C are not yet supported. We plan to incorporate theories of finite precision integers and floating-point numbers soon. Of course, the verifier is incomplete: the theorem prover could fail to prove a valid assertion, and the state space could be infinite (or too large) even after the symbolic abstraction. However the language does provide flexible mechanisms for specifying a finite subset of the state space, e.g., by bounding inputs with assumptions such as `$assume 1<=N && N<=B`, where `B` can be specified on the command line.

Experiments. The examples include many concurrency staples (e.g., variations on dining philosophers), as well as models of concurrent programs from the literature using the APIs discussed above. A sample of these, and the results of applying the CIVL verifier, follow. The experiments were run on a Mac Mini with a 2.6 GHz Intel Core i7 and 4 GB RAM. The data include the time in seconds, the number of atomic transitions executed in the search (steps) and the number of calls to CVC3.

- `ring.c` [31] (MPI): n processes exchange data cyclically m times; verified for $1 \leq n \leq 10$, $1 \leq m \leq 5$, 5.5s, 11761 steps, 477 prover calls.
- `diffusion1d_par.c` [30] (MPI): solves 1d-diffusion equation with n processes, k time steps, m points; verified functional equivalence with sequential version for $1 \leq m, n \leq 6$, $k = 3$, 15.9s, 13673 steps, 986 prover calls.
- `fig4.98-threadprivate.c` [8] (OpenMP): example using thread private pragma and parallel for loop over array of length m with n threads; verified for $1 \leq m \leq 12$, $1 \leq n \leq 10$, 17.1s, 247854 steps, 418 prover calls.
- `dot.cu` [27, Chap. 5] (CUDA): computes dot product of two vectors of length n with m threads per block and $\lceil n/m \rceil$ blocks; verified for $1 \leq n \leq 12$ and $m = 4$, 11.9s, 130967 steps, 64 prover calls.
- `dining`: deadlock-free version of n dining philosophers; verified for $2 \leq n \leq 8$, 5.6s, 93673 steps, 32 prover calls.
- `mpi-pthreads.c` [4] (MPI/Pthreads): two processes each spawn two threads which send and receive a sequence of messages. Derived from a defect report submitted to MPICH, this code contains a known defect leading to deadlock revealed only with a subtle interleaving. CIVL produced a minimal counterexample in 1.64s, 2309 steps, 2 prover calls.

3 Related Work

Model checking. Many model checking tools specialize in a specific concurrency API or language: e.g., TASS and ISP [19] (C/MPI), Java PathFinder [25] (Java), DiVinE 3.0 [5] (C/Pthreads), and CVT [34] (Chapel).

CIVL's guarded command language is similar to Promela, the language of the model checker Spin [22]. Promela lacks procedures and any way to nest process definitions, so for most purposes there is only a "global" (shared) scope, and one "local" scope for each process. The dSPIN model checker added many dynamic constructs to Promela, including procedures, arbitrary nesting of scopes within a procedure, and heap-allocated data [12]. However, dSPIN procedures can only be declared in the global scope, which is therefore the only scope which may contain data shared by multiple processes. This contrasts with the arbitrary hierarchical sharing which is fundamental to CIVL.

Intermediate representations. The intermediate representation used by the Bandera model checking platform used a similar guarded transition system representation and supported an extensible type system [10]. Zing [2, 33] is another verification language with a rich type system (including a *set* type), support for object-oriented features, heap allocation, and spawning of processes. Boogie [23] and Why3 [14] are examples of a new generation of intermediate verification languages aimed primarily at the application of automated theorem-proving approaches to sequential programs; Boogie has recently added limited support for concurrency. None of these languages allows arbitrary nesting of scopes or the use of different scopes shared by subsets of processes.

Programming languages. Some CIVL features are of course found in programming languages. While C does not allow nested procedures, the GCC extension of C does [32, §6.4]. UNIX introduced the C `fork` and `wait` procedures [26], the building blocks of "unstructured parallelism" which have been re-used in numerous contexts. Unlike CIVL's `spawn`, UNIX `fork` creates an entire new copy of a process (including the call stack), so there is no sharing of scopes (instead, message-passing is used). Cilk [17] and Erlang [3] also provide `spawn` primitives, but don't allow nested procedures. One language which does provide all these features is Racket [15], and in theory it should be possible to represent a CIVL-C program in Racket (perhaps ignoring pointers).

Cyclone adds to C a notion of "regions" together with qualifiers restricting the region which a pointer may reference [20]. Regions are similar in some respects to scopes and we are exploring a scope-modified pointer type in CIVL. Other recent language designs targeting parallel execution have adopted the notions of regions. Phalanx [18] uses a "place" hierarchy which is similar in some respects to CIVL scopes and a form of region-based pointer categorization. ParaSail [24] also uses region-based memory management where regions are associated with stack frames and programmers can express lifetime relationships among objects to control the region they reside in.

Acknowledgment. Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1346769 and CCF-1346756.

References

1. ABC: ANTLR-Based C front-end. http://vsl.cis.udel.edu/abc (accessed Feb 6, 2014)
2. Andrews, T., Qadeer, S., Rajamani, S.K., Rehof, J., Xie, Y.: Zing: Exploiting program structure for model checking concurrent software. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004 - Concurrency Theory. Lecture Notes in Computer Science, vol. 3170, pp. 1–15. Springer Berlin Heidelberg (2004), http://dx.doi.org/10.1007/978-3-540-28644-8_1
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall Europe, Herfordshire, UK, second edn. (1996)
4. Balaji, P., Dinan, J., Hoefler, T., Thakur, R.: Advanced MPI programming. Tutorial at SC13: International Conference on High Performance Computing, Networking, Storage, and Analysis, Denver, Colorado, November 2013, http://www.mcs.anl.gov/~thakur/sc13-mpi-tutorial/
5. Barnat, J., Brim, L., Havel, V., Havlícek, J., Kriho, J., Lenco, M., Rockai, P., Still, V., Weiser, J.: DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 863–868. Springer (2013)
6. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4590, pp. 298–302. Springer (2007)
7. Board, O.A.R.: OpenMP API Specification for Parallel Programming. http://openmp.org/wp/, accessed Feb. 8, 2014
8. Chapman, B., Jost, G., van der Pas, R.: Using OpenMP: Portable Shared Memory Parallel Programming. MIT Press, Cambridge, Massachusetts (2008), http://openmp.org/examples/Using-OpenMP-Examples-Distr.zip, (examples)
9. clj-ds: Clojure's data structures modified for use outside of Clojure. https://github.com/krukow/clj-ds (accessed Feb 6, 2014)
10. Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby, Zheng, H.: Bandera: Extracting finite-state models from Java source code. In: ICSE '00: Proceedings of the 22nd International Conference on Software Engineering. pp. 439–448. ACM, New York, NY, USA (2000)
11. CUDA Programming Guide Version 5.0. http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed Feb. 8, 2014
12. Demartini, C., Iosif, R., Sisto, R.: dSPIN: A dynamic extension of SPIN. In: Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking. pp. 261–276. Springer-Verlag, London, UK (1999), http://dl.acm.org/citation.cfm?id=645879.672057
13. Dwyer, M.B., Hatcliff, J., Robby, Ranganath, V.P.: Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. Form. Methods Syst. Des. 25, 199–240 (September 2004)
14. Filliâtre, J.C., Paskevich, A.: Why3: Where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) Proceedings of the 22nd European Conference on Programming Languages and Systems. pp. 125–128. ESOP'13, Springer-Verlag, Berlin, Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-37036-6_8

15. Flatt, M., PLT: The Racket reference, version 5.3.1. http://docs.racket-lang.org/reference/, retrieved Nov. 19, 2012
16. Forum, M.P.I.: MPI: A message-passing interface standard, version 3.0. http://www.mpi-forum.org/docs/docs.html (Sep 2012)
17. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI). pp. 212–223. Montreal, Quebec, Canada (Jun 1998), proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.
18. Garland, M., Kudlur, M., Zheng, Y.: Designing a unified programming model for heterogeneous machines. In: SC '12: Proc. Conference on High Performance Computing Networking, Storage and Analysis (Nov 2012), to appear
19. Gopalakrishnan, G., Kirby, R.M., Siegel, S., Thakur, R., Gropp, W., Lusk, E., De Supinski, B.R., Schulz, M., Bronevetsky, G.: Formal analysis of MPI-based parallel programs. Communications of the ACM 54(12), 82–91 (Dec 2011), http://doi.acm.org/10.1145/2043174.2043194
20. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation. pp. 282–293. PLDI '02, ACM, New York, NY, USA (2002), http://doi.acm.org/10.1145/512529.512563
21. Group, K.: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/
22. Holzmann, G.J.: The Spin Model Checker. Addison-Wesley, Boston (2004)
23. Leino, K.R.M.: This is Boogie 2. http://research.microsoft.com/apps/pubs/default.aspx?id=147643 (Jun 2008)
24. ParaSail Programming Language. http://www.parasail-lang.org (accessed Feb 7, 2014)
25. Pasareanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) ASE. pp. 179–180. ACM (2010)
26. Ritchie, D.M., Thompson, K.: The UNIX time-sharing system. Commun. ACM 17(7), 365–375 (Jul 1974), http://doi.acm.org/10.1145/361011.361061
27. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley (2010), https://developer.nvidia.com/content/cuda-example-introduction-general-purpose-gpu-programming-0
28. SARL: The Symbolic Algebra and Reasoning Library. http://vsl.cis.udel.edu/sarl (accessed Feb 6, 2014)
29. Siegel, S.F., Mironova, A., Avrunin, G.S., Clarke, L.A.: Combining symbolic execution with model checking to verify parallel numerical programs. ACM Transactions on Software Engineering and Methodology 17(2), Article 10, 1–34 (2008)
30. Siegel, S.F., Zirkel, T.K.: FEVS: A Functional Equivalence Verification Suite for high performance scientific computing. Mathematics in Computer Science 5(4), 427–435 (2011)
31. Siegel, S.F., Zirkel, T.K.: Loop invariant symbolic execution for parallel programs. In: Kuncak, V., Rybalchenko, A. (eds.) Verification, Model Checking, and Abstract Interpretation: 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22–24, 2012, Proceedings. LNCS, vol. 7148, pp. 412–427. Springer (2012)

32. Stallman, R.M., the GCC Developer Community: Using the GNU Compiler Collection: For GCC version 4.7.2. GNU Press, a division of the Free Software Foundation (2010), http://gcc.gnu.org/onlinedocs/gcc, http://gcc.gnu.org/onlinedocs/gcc
33. Zing language specification, Microsoft Corporation. http://research.microsoft.com/en-us/projects/zing/zinglanguagespecification.pdf (2005)
34. Zirkel, T.K., Siegel, S.F., McClory, T.: Automated verification of Chapel programs using model checking and symbolic execution. In: Brat, G., Rungta, N., Venet, A. (eds.) NASA Formal Methods: 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7871, pp. 198–212. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-38088-4_14