CIVL: Applying a General Concurrency Verification Framework to C/Pthreads Programs (Competition Contribution)

Manchun Zheng[1](✉), John G. Edenhofner[1], Ziqing Luo[1], Mitchell J. Gerrard[2], Michael S. Rogers[2], Matthew B. Dwyer[2], and Stephen F. Siegel[1]

[1] Department of Computer and Information Sciences, University of Delaware, Newark, USA
{zmanchun,johneden,ziqing,siege}@udel.edu

[2] Department of Computer Science and Engineering, University of Nebraska, Lincoln, USA
{mgerrard,mrogers,dwyer}@cse.unl.edu

Abstract. CIVL is a framework for the analysis and verification of concurrent programs. The front-end translates C programs that use (subsets of) Pthreads, MPI, OpenMP, or CUDA—alone or in combination—to an intermediate verification language CIVL-C. The back-end uses symbolic execution and model checking techniques to verify a number of safety properties of a CIVL-C program, such as absence of assertion violations, deadlocks, or out-of-bound indexes. We submit CIVL for verifying Pthreads programs in the concurrency category.

1 Verification Approach

CIVL [8] is a framework for verifying parallel programs written using various concurrency libraries or language extensions such as MPI [3], POSIX threads ("Pthreads") [2], OpenMP [6], and CUDA [5]. (Significant subsets of each of these concurrency "dialects" is supported; CUDA support excludes C++ features.) CIVL compiles programs to the CIVL-C modeling language, which extends sequential C11 with concurrency and verification primitives and linguistic features, such as nested functions and scoped memory. For each dialect, an AST "transformer" and libraries are used to express the original program as an equivalent CIVL-C program. Different transformers can work together to convert programs using multiple dialects into CIVL-C. [1]

CIVL uses a combination of explicit model checking and symbolic execution for verification. Model checking is used to explore the thread and process interleavings introduced by a concurrency model. CIVL uses state-of-the-art partial order reduction to mitigate the state space explosion problem. Symbolic execution further reduces the state space by collapsing sets of equivalent values along

[1] Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1319571, CCF-1346769 and CCF-0953210.

M. Chechik and J.-F. Raskin (Eds.): TACAS 2016, LNCS 9636, pp. 908–911, 2016.
DOI: 10.1007/978-3-662-49674-9_57

program executions. CIVL makes use of the Symbolic Analysis and Reasoning Library (SARL) [7] which is a package for normalizing, caching, and determining validity queries over logical formulae. SARL can leverage multiple Satisfiability Modulo Theories (SMT) solvers, but in general more than 99.5 % of the queries generated in a verification run are solved within SARL and do not require invocation of an SMT solver [8].

2 Software Architecture

The CIVL framework (Fig. 1) is distributed as open source software under the GNU General Public License and consists of several components. ABC is a C11 front-end which generates Abstract Syntax Trees (AST) from CIVL-C programs. The CIVL back-end builds a state-transition model based on the AST, then uses GMC (Generic Model Checker) and SARL to perform model checking and to manipulate symbolic state encodings to compute next states. For the competition, two theorem provers are used: CVC4 [1] and Z3 [4].

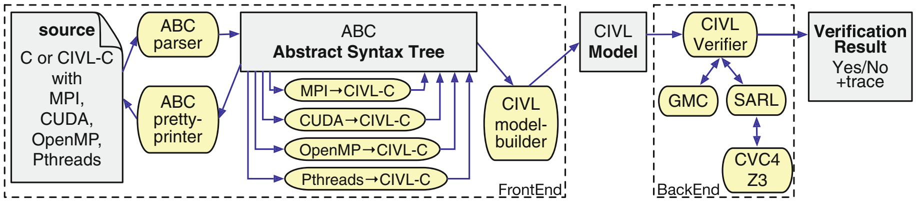

Fig. 1. The CIVL project architecture

CIVL is implemented in Java 7. It comes equipped with pre-built libraries to model system functions and concurrent data structures to support a variety of process and thread-level concurrency models. These libraries allow new concurrency dialects to be supported directly in the CIVL-C language, which reduces the cost of extending CIVL.

3 Strengths and Weaknesses

The most significant strength of CIVL is its ability to verify programs that use a variety of concurrency dialects, including "hybrid" programs that use multiple dialects, such as MPI+Pthreads. CIVL also checks a large number of generic properties, including absence of divisions by zero, reads of uninitialized variables, and out-of-bound array indexing. In fact CIVL found defects of each of these kinds in the SV-COMP suite; these defects were subsequently corrected. Additional properties include absence of memory leaks and illegal pointer dereferences, and dialect-specific properties, such as absence of "potential deadlocks" in MPI programs. In addition, CIVL can verify the functional equivalence of two versions of a C program with one or multiple of the four concurrency dialects, such as a trusted sequential version and a more complicated parallel one.

The CIVL back-end (verifier) suffers from the state explosion problem, and scalability can become an issue for programs that access shared variables frequently or have many nondeterministic choices. For the competition, small bounds were placed on the number of live threads (6). A "downscaling" transformation is performed that replaces array lengths above a certain threshold (11) with a small number (3); a similar transformation is applied to the upper bounds in `for` loops. These are unsound transformations, but nevertheless allowed CIVL to obtain the expected result for all of the examples in the concurrency category.

4 Setup and Configuration

CIVL v1.5 (available at http://vsl.cis.udel.edu/civl/svcomp16) is used for SVCOMP 2016. CIVL is distributed as a single jar file, which can be placed in any readable directory. Then an executable file named `civl` should be created and placed in the `PATH`; this file has the form

```
#!/bin/sh
java -Xmx15000M -Duser.home=$HOME -Djava.io.tmpdir=$TMPDIR \
  -jar /path/to/civl.jar $@
```

The executables `java` (a Java $\geq$7 VM), `cvc4` (version 1.4), and `z3` (version $\geq$4.3.2) must also be in the `PATH`. Finally, the command "`civl config`" should be executed once. This will search for appropriate theorem provers in the `PATH` and create a file named `.sarl` in the user's home directory containing information about each. The entries for CVC4 and Z3 should appear in that file.

CIVL is submitted for the concurrency category of the competition. The option `-svcomp16` is used, which bundles the type and process bounds described above. The command for the competition is `civl verify -svcomp16 source.i`, where `source.i` is the file name of a target program. The wrapper script `civl.py` can be used to interpret verification results.

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
2. IEEE: Portable Operating System Interface (POSIX) Base Specifications, IEEE Std 1003.1-2008, 2013 ed (2013). http://www.unix.org/version4/
3. Message-Passing Interface Forum: MPI: A Message-Passing Interface standard, version 3.0. http://www.mpi-forum.org/docs/docs.html
4. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. NVIDIA: CUDA C Programming Guide Version 7.5. http://docs.nvidia.com/cuda/cuda-c-programming-guide/. Accessed 31 Oct 2015

6. OpenMP Architecture Review Board: OpenMP API Specification for Parallel Programming. http://openmp.org/wp/. Accessed 8 Feb 2015
7. SARL: The Symbolic Algebra and Reasoning Library. http://vsl.cis.udel.edu/sarl. Accessed 6 Feb 2015
8. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: the concurrency intermediate verification language. In: SC15 (2015). http://doi.acm.org/10.1145/2807591.2807635