CIVL: Formal Verification of Parallel Programs

Manchun Zheng*, Michael S. Rogers†, Ziqing Luo*, Matthew B. Dwyer†, and Stephen F. Siegel*
*Department of Computer and Information Sciences, University of Delaware, USA Email: zmanchun,ziqing,siegel@udel.edu
†Department of Computer Science and Engineering, University of Nebraska - Lincoln, USA Email: mrogers,dwyer@cse.unl.edu

***Abstract*—CIVL is a framework for static analysis and verification of concurrent programs. One of the main challenges to practical application of these techniques is the large number of ways to express concurrency: MPI, OpenMP, CUDA, and Pthreads, for example, are just a few of many "concurrency dialects" in wide use today. These dialects are constantly evolving and it is increasingly common to use several of them in a single "hybrid" program. CIVL addresses these problems by providing a *concurrency intermediate verification language*, CIVL-C, as well as translators that consume C programs using these dialects and produce CIVL-C. Analysis and verification tools which operate on CIVL-C can then be applied easily to a wide variety of concurrent C programs. We demonstrate CIVL's error detection and verification capabilities on (1) an MPI+OpenMP program that estimates π and contains a subtle race condition; and (2) an MPI-based 1d-wave simulator that fails to conform to a simple sequential implementation.**

I. Introduction

Parallel programming is notoriously difficult. When a defect manifests itself only for some specific scheduling order, it is hard to reproduce, isolate, and repair the problem using traditional debugging or testing techniques. A number of new approaches have been proposed to address this problem, including model checking, symbolic execution, and various kinds of static analysis. These approaches have been shown to be effective in some specific contexts.

One of the main challenges to pushing the field forward is the sheer number of different ways to express parallelism. A large number of concurrency APIs, libraries, language extensions, and entirely new languages have been introduced. Some of the most commonly used are the message passing library MPI [1], the thread-based library POSIX threads (Pthreads) [2], the pragma-based annotation system OpenMP [3], and the GPU-based language extension CUDA [4]. We refer to all of these mechanisms as "concurrency dialects".

The authors would thank Timothy K. Zirkel, Andre V. Marianiello, and John G. Edenhofner for their contribution to the development of CIVL. Funding for the CIVL project is provided by the U.S. National Science Foundation under awards CCF-1319571, CCF-1346769 and CCF-0953210. This material is also based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number DE-SC0012566. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

The concurrency dialects are constantly changing, as more features are added with each update of the standards. For example, there are three versions of the MPI specification within 8 years [1]. New dialects are introduced regularly. And it is increasingly common for programmers to combine multiple dialects in a single "hybrid" parallel program [5], [6].

All of this creates a nightmare for those seeking to develop practical analysis or verification tools for parallel programs. The tools usually target one very small piece of the concurrency landscape; they go out of date as standards evolve; they cannot be combined to apply to hybrid programs. There is also significant duplication of effort, as the same basic techniques are re-implemented for different dialects.

The CIVL framework attempts to address these problems by providing a common *concurrency intermediate verification language*. C programs that use the concurrency dialects are translated into the intermediate language, CIVL-C. Verification and analysis tools operate on CIVL-C. The idea is that when a concurrency dialect is introduced, or an old one changes, only a front-end translator needs to be updated. Dually, a new analysis technique can be implemented on CIVL-C, and immediately applied to programs using any of the dialects.

Fig. 1 presents an overview of the CIVL framework. CIVL uses the ABC compiler front-end [7] to generate an AST from the source code file(s). ABC recognizes programs written in the C11 dialect of C, CUDA-C, and CIVL-C, and supports analyses and transformations on the AST level. For each of the four concurrency dialects, a transformer replaces dialect-specific code with semantically equivalent CIVL-C code. The transformers can be applied in sequence to translate hybrid programs. The result is a "pure" CIVL-C AST.

The back-end contains the CIVL model builder and verifier. A CIVL model is a lower-level representation akin to a program graph. The verifier performs an explicit enumeration of the reachable states of the model ("model checking") through a generic DFS algorithm. Static analysis over memory access through pointers is applied, based on which partial order reduction is conducted to eliminate unnecessary interleavings [8]. In each state, the values assigned to variables are symbolic expressions ("symbolic execution"). CIVL uses the Symbolic Algebra and Reasoning Library (SARL) [9], to manipulate and reason about symbolic expressions. SARL in turn makes use of state-of-the-art theorem provers, CVC3 [10], CVC4 [11], and Z3 [12], to resolve validity/satisfiability queries. At each state, CIVL checks a number of *standard properties*: absence of assertion violations, deadlocks, memory leaks, improper pointer dereferences or arithmetic, out-of-bound array indexes, and division by zero. The assertions may be in the original source code, or they may have been inserted by a transformer. A number of dialect-specific properties are verified by inserting

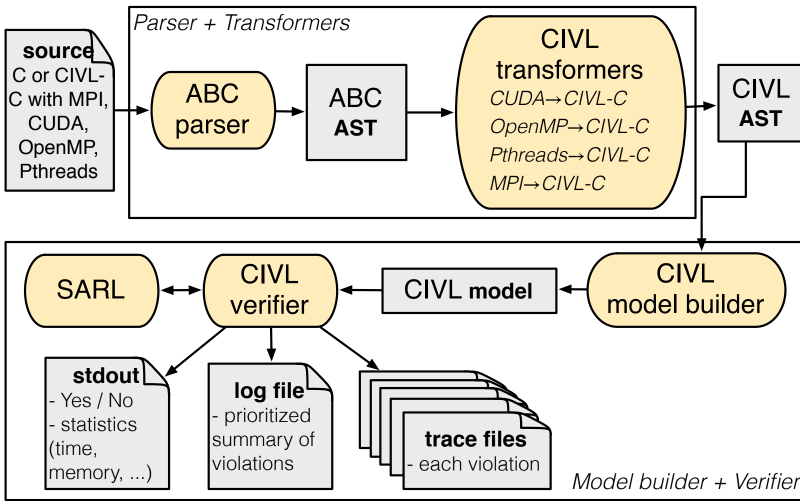

Fig. 1. The CIVL framework

`$input`: type qualifier for input variables
`$output`: type qualifier for output variables
`$assume(expr)`: assumptions added to path condition
`$assert(expr)`: assertions to be checked
`$scope`: type for reference to a dyscope
`$proc`: type representing reference to an executing process
`$malloc(scope,size)` :
 allocates object in heap of specified dyscope
`$free(ptr)` : deallocates object in heap
`$choose_int(n)`:
 nondeterministically returns an integer in $[0, n-1]$
`$choose {stmt1 stmt2 ... default: stmt}`:
 nondeterministic selects one enabled statement
`$spawn f(arg0, ...)`: creates a process executing `f`
`$wait(p)` : blocks until process p terminates
`$waitall(procs, n)`: blocks until n processes terminate
`$parfor(int i,j,...: d) stmt` :
 spawns processes for `d` and waits until all terminate
`$when(guard) stmt` :
 blocks until `guard` evaluates to *true*
`$atomic stmt` : executes `stmt` as one trace step

Fig. 2. Some commonly-used CIVL-C primitives

appropriate assertions, e.g., that accesses to OpenMP shared variables avoid race conditions as specified by OpenMP's weak memory consistency model.

CIVL has been used to verify over 100 parallel programs written in different concurrency dialects, including OpenMP, MPI, Pthreads, CUDA and hybrid programs of MPI+OpenMP or MPI+Pthreads. These programs have been collected from various communities, e.g., the LLNL OpenMP online tutorial exercises [13], the SV-COMP Pthreads concurrency benchmarks [14], the FEVS suite of MPI benchmarks [15]. CIVL is described in detail in [8], along with the results for verification of 36 examples. The entire framework can be downloaded in source code or as a single JAR file from the CIVL web page, http://vsl.cis.udel.edu/civl. Results and artifacts from a large case study, consisting of over 366 files and 27k SLOC (including 14 examples with 200+ SLOC and over 90 with 100+ SLOC) are also available on the web site. Multiple large case studies [16] (10s of thousands SLOC) are in progress, but open source applications of intermediate size using the dialects that CIVL targets have been difficult to find.

There are many possible applications of the framework, but in this demonstration we focus on two scenarios. In the first, CIVL is used to verify the standard properties (including dialect-specific assertions) in a hybrid MPI+OpenMP program that computes an estimate of π. In the second, CIVL checks the functional equivalence of two programs that compute a solution to 1-dimensional wave equation, one a simple sequential implementation, the other a much more complicated MPI version. In each scenario, CIVL finds a bug and produces a minimal counterexample; after fixing the bug, CIVL is applied again to confirm the fix. More information about the demonstration can be found at http://vsl.cis.udel.edu/civl/ase15.

II. CIVL IN PRACTICE

The modeling language of CIVL, CIVL-C, extends the sequential part of C11 with new primitives for concurrency and specification, various abstract datatypes, and the ability to define functions in any scope, among other things. Concurrnecy primitives provide the means for the creation, termination, and communication between processes. All CIVL-C keywords start with the symbol `$`, so that identifiers in strictly conforming C programs will never conflict with a CIVL-C keyword. We list some of the CIVL-C primitives in Fig. 2. An input variable is a read-only global variable and is initialized with an unconstrained value of its type; an output variable is a write-only global variable. There are a number of built-in constants, like `$here`, of `$scope` type, evaluating to the reference to the current *dynamic scope* (or *dyscope*; see below); and `$self`, of `$proc` type, evaluating to the reference to the current process. For a complete description of the CIVL-C language, see the CIVL manual [17].

The verifier uses symbolic techniques to explore the state space of a CIVL model. A state consists of a path condition, a tree of dyscopes, and a set of processes. A dyscope is the dynamic evaluation of a lexical scope, and assigns values to each variable declared in the lexical scope. Each dyscope contains a heap for allocating memory dynamically. A process is represented by a stack of call frames, each of which specifies the current program location and a dyscope. These are highly dynamic structures: new dyscopes are born whenever control enters a new lexical scope and disappear when control leaves the scope; processes are born whenever a `$spawn` or similar statement is executed and disappear when the spawned function returns. A transition in the state space corresponds to the execution of a single atomic statement by one process. The CIVL verifier also uses a multitude of reduction techniques to reduce the number of states that need to be explored. For example, it detects "purely local" transitions automatically and executes a sequence of such transitions in a single process greedily, grouping them into a single *trace step*.

CIVL uses a command-line interface providing four basic commands: `run`, simulating the execution of a program by randomly selecting one branch for each nondeterminstic choice; `verify`, checking a program exhaustively for properties mentioned above; `compare`, in addition to properties checked by `verify`, checking two programs for functional equivalence; and `replay`, reproducing a counterexample when there is a violation. When CIVL detects a violation, it records it in a log, and then continues the checking until an "error bound" is reached or the search ends. As shown in Fig. 1, the output of CIVL is three-fold. First, it prints a summary of the verification task to standard out, including

whether any violations are detected, a summary of each violation, and statistics such as time and memory used, number of states explored, number of transitions executed, and the number of theorem prover queries. If violations are detected, a log file is generated which categorizes the violations and selects a representative of minimal trace length from each category. Further, a trace file is created for each representative and is later used by `replay` to reproduce the counterexample.

In addition, there are a number of options to be configured, for printing extra information like states, transitions, path conditions, and so on. Moreover, CIVL provides an option *min* for finding a minimal counterexample for each violation, which is extremely helpful for diagnosing and repairing defects.

CIVL models a substantial subset of the C11 standard libraries, such as `stdio`, `stdlib`, `string`, `math` and so on. For each function prototype, CIVL either provides a definition for it in CIVL-C code, or models its behavior in Java code as a library component. In order to verify a program using third-party libraries, one would need to model the implementation of those libraries in CIVL-C code and/or add a library component for behaviors beyond CIVL-C like comparing the value of two objects of arbitrary type through their pointers.

CIVL, ABC, and SARL are open-source and implemented in Java 7. The implementation of CIVL contains over 10 modules, which provide interfaces for various purposes such as AST transformation, symbolic evaluation of expressions, execution of statements, manipulation of states, implementation of libraries and so on. The APIs are designed with a clear structure and are well documented. Moreover, the developers follow certain standards, such as maintaining an acyclic "uses" relation among modules, to keep the design understandable and adaptable. The design allows developers with different needs to work on CIVL at the same time. For example, a professor would like to design an algorithm for the independence analysis of loops for OpenMP programs at the AST level, and he can implement this in CIVL by creating a transformer class that implements a generic transformer interface. Meanwhile, a graduate student can work on another transformer class to translate OpenMP programs into CIVL-C, without interfering with the work of the professor. To support maintenance, a test suite is configured to run automatically after each commit to the code base, and new tests are added to the suite for new features or in response to defect reports. Code coverage is also analyzed at each commit, and the results of the tests and the coverage analysis are sent to a publicly viewable web page. In addition, there is a Trac system for users to report bugs, which are generally resolved in a few days. New versions of CIVL are released regularly; the current version is 1.4, which is the 23rd released version.

III. VERIFYING A HYBRID PARALLEL PROGRAM

Fig. 3 shows a sample MPI+OpenMP program, which contains a parallel loop with an `MPI_Send` routine call. The OpenMP transformer translates the parallel loop into a `$parfor` construct, and introduces new variables for each shared variable and uses helper functions for accessing them. Applying the OpenMP transformation to the code in Fig. 3 results in the code in Fig. 4. The thread team of a parallel region is modeled by the shared variable `gt`, which has type

```
1 ⟨external-definitions⟩
2 int main() {
3   int a[N]; ...
4   #pragma omp parallel shared(a)
5   #pragma omp for
6   for (int i=0; i<N; i++) {
7     a[i] = i;
8     MPI_Send(&a[i], ...);
9   }...
10 }
```

Fig. 3. MPI+OpenMP code sample

```
1 ⟨external-definitions⟩
2 int main() {
3   int a[N]; ...
4   int nth = 1 + $choose_int($omp_numThreads);
5   $omp_gteam gt = $omp_gteam_create($here, nth);
6   $omp_gshared a_gs = $omp_gshared_create(gt,&a);
7   $parfor (int _tid : 0 .. nth-1) {
8     $omp_team t = $omp_team_create($here,gt,_tid);
9     int a_l[N]; int a_st[N];
10    $omp_shared a_s=
11         $omp_shared_create(t,a_gs,&a_l,&a_st);
12    $domain(1) l_d=($domain){0 .. N - 1 # 1};
13    $domain(1) iters= $omp_arrive_loop(t,0,l_d,2);
14    $for (int i : iters) {
15      double tmpWrite0 = i;
16      $omp_write(a_s, &a_l[i], &tmpWrite0);
17      int tmpRead0;
18      $omp_read(a_s, &tmpRead0, &a_l[i]);
19      MPI_Send(&tmpRead0, ...);
20    }
21    $omp_barrier_and_flush(t);
22    $omp_shared_destroy(a_s);
23    $omp_team_destroy(t);
24  }
25  $omp_gshared_destroy(a_gs);
26  $omp_gteam_destroy(gt); ...
27 }
```

Fig. 4. Code in Fig. 3 after OpenMP transformation

`$omp_gteam`. This is a pointer to a structure which maintains the state of each thread and each shared variable. Each thread has a local object, `t`, which contains its thread ID and a reference to `gt`. Each shared variable, e.g., `a`, is modeled using several data structures. These include two variables shared by all threads in the team: `a`, which represents the "global view" of the shared variable, and `a_gs`, which contains references to `gt` and `a`. Three thread-local variables are introduced: `a_l`, which represents the thread's "local view" of `a`; `a_st`, which maintains the "status" of the thread's local view, e.g., whether it has been modified by the thread since the last flush; and `a_s`, which wraps together references to `a_l` and `a_st` and

```
1 $input int _mpi_np, _mpi_np_lo, _mpi_np_hi;
2 $assume(_mpi_np_lo <= _mpi_np && _mpi_np <= _mpi_np_hi);
3 CMPI_Gcomm _mpi_gc = CMPI_Gcomm_create($here, _mpi_np);
4 void _mpi_proc(int _mpi_rank){
5   MPI_Comm MPI_COMM_WORLD =
6          CMPI_Comm_create($here, _mpi_gc, _mpi_rank);
7   ⟨external-definitions⟩
8   int _mpi_main(){
9     int a[N]; ...
10    int nth = 1 + $choose_int($omp_numThreads); ...
11    MPI_Send(&tmpRead0, ...); ...
12  }
13  _mpi_main();
14  CMPI_Comm_destroy(MPI_COMM_WORLD);
15 }
16 int main(){
17   $parfor(int i: 0 .. _mpi_np-1) _mpi_proc(i);
18   CMPI_Gcomm_destroy(_mpi_gc);
19 }
```

Fig. 5. Code in Fig. 4 after MPI+OpenMP transformation

```
1 int main(){...
2   double PI24 = 3.141592653589793238462643; ...
3   MPI_Bcast(&nitr, 1, MPI_INT, Root, ...); ...
4   MPI_Bcast(&ScatterSize, 1, MPI_INT, Root, ...); ...
5   #pragma omp parallel for private(itr)
6       reduction(+: psum)
7   for (itr = start; itr <= end; itr++){
8     x = h * ((double) itr - 0.5);
9     psum = psum + func(x);
10  }
11  MPI_Reduce(&mypi, &pi, 1, ...); ...
12  assert(fabs(PI24-pi) < ERR); ...
13 }
```

Fig. 6. π calculation in MPI+OpenMP

is used as the argument to the functions which access the shared variable, i.e., `$omp_write` and `$omp_read`. E.g., the assignment to `a[i]` (Fig. 3, line 7) is translated to an `$omp_write` call (Fig. 4, line 16). Similarly, reading `a[i]` (Fig. 3, line 8) is modeled by `$omp_read` (Fig. 4, line 18). This scheme enables precise detection of errors related to the OpenMP memory model. If a second thread that attempts to write to the same subcomponent of a shared variable without an intervening flush, a data race will be reported.

With the MPI transformer applied to Fig. 4, the final translation result turns into that in Fig. 5. The code before MPI transformation, i.e., the result of the OpenMP transformation, is wrapped in the function `_mpi_proc`. The original main function is renamed as `_mpi_main`, and defined in the scope of the function `_mpi_proc`. In this way, the original global declarations all become "local" for each MPI process. The main function of the final program contains a `$parfor` construct for spawning processes to execute `_mpi_proc` with different ranks and waits until they all terminate. In addition, there is a global *communicator* `_mpi_gc`, shared by all processes, which stores the buffered messages. Each process has its local *communicator handle* `MPI_COMM_WORLD`, which is essentially a reference to `_mpi_gc`. Input variables `_mpi_np`, `_mpi_np_lo` and `_mpi_np_hi` are introduced to specify the number of MPI processes, where `_mpi_np_lo` and `_mpi_np_hi` stand for the lower and upper bound of the number of processes `_mpi_np`, respectively.

We use CIVL to verify an MPI+OpenMP program that calculates π [18], an excerpt of which is shown in Fig. 6. The computation task includes a number of loop iterations, which are partitioned among the MPI processes. For each MPI process, its assigned task is executed in an OpenMP for loop in a parallel region. In the original program, the variable `x` is declared in the `private` list at line 5. In our example, we purposely remove it in order to introduce a race condition. With the number of MPI processes and OpenMP threads both set to 2, CIVL is able to detect the race condition in a few seconds on a personal laptop. The report states "Thread 0 can't perform `$omp_write` because thread 1 has written to the same variable and hasn't flushed yet." With the option *min* enabled, CIVL reports a minimal counterexample of 586 steps and replays it. After adding the variable *x* back to the `private` list, CIVL verifies the program successfully without reporting any violations.

IV. Checking Functional Equivalence

During the process of software development, there may arise different versions of a software component or an algorithm. For example, one might choose to reimplement a sequential application to take advantage of multi-core CPU/GPU hardware. These different implementations are often expected to be functionally equivalent to each other.

CIVL is often able to prove the functional equivalence of two programs, i.e., given the same input, both programs will produce the same output. (As always, this is within specified bounds on process counts, array lengths, etc.) Either or both programs may be sequential or parallel and may use any combination of the four dialects. As shown in Fig. 7, two programs, referred to as *spec* (specification) and *impl* (implementation) are parsed and transformed separately, producing two CIVL-C ASTs. Then the *compare combiner* is applied, yielding a single AST that runs *spec* and *impl* in sequence and compares their output using a number of assertions, as illustrated in Fig. 8. The back-end is then used to generate and verify the model as usual. If there exists nondeterminism in the specification, then CIVL would have redundant explorations of the implementation. However, this could be resolved through moderate modification, i.e., making CIVL explore the specification and the implementation separately and record their outputs and the corresponding path conditions, and then compare the recorded outputs to see if given the same path condition, both runs always result in the same output.

We use CIVL to compare two implementations of a 1d-wave equation solver. These determine the movement of a wave through a string with fixed boundary points. The first, *wave1d_seq.c* is a simple sequential C program used as the specification. The second, *wave1d.c*, is a much more complicated MPI version. Both are included in the FEVS suite [15], [19]. Fig. 9 presents an excerpt of *wave1d.c*. Some additional CIVL-C code has been inserted to specify the intended behavior of the program; in the full source, this extra code is within preprocessor directives `#ifdef _CIVL ... #endif` which cause the extra code to be ignored during normal compilation but used by CIVL. The program starts with the string in some arbitrary initial position specified by the input variable `u_init`. The two programs are compared by CIVL with the upper bound of both the size of the initial vector and the number of time steps being 5. For the MPI program, the number of processes is restricted between 1 and 4 (inclusive).

During the comparison, CIVL finds a violation after 630 trace steps, where the output vector of *wave1d_seq.c* disagrees with that of *wave1d.c*. With the *min* option enabled, CIVL continues to explore the state space until it finds the minimal violation with only 193 trace steps, which is the case when the lower bounds of the inputs are chosen. Fig. 10 is a sample of the trace printed by CIVL `replay`. For example, step 110 is performed by process p1 at state 118. It contains one transition of a *malloc* statement, which corresponds to line 10 in Fig. 9, and emanates from program location 309. When the transition completes, the control of p1 goes to program location 310 and the current state becomes state 120. The step also displays the exact value of each expression of each statement being executed. For example, step 153 indicates that the value of `&u_curr[1]` (line 14 in Fig. 9) is `&<d21>heap.malloc4[0][1]`, which means the address of the second element of the first heap object in dyscope 21, which is created by the fourth `$malloc` of the program.

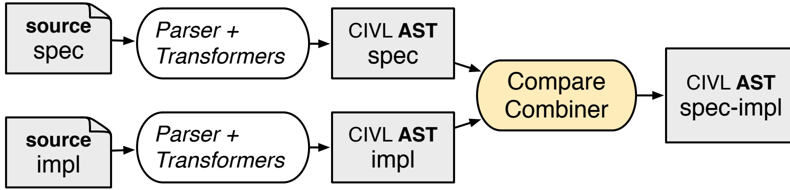

Fig. 7. CIVL for functional equivalence

```
1 $input ... input0, input1, ...;
2 $output ... out0$spec, out1$spec, ...;
3 $output ... out0, out1, ...;
4 void specification(){⟨orignal code ...⟩}
5 void implementation(){⟨orignal code ...⟩}
6 int main(){
7   specification(); implementation();
8   $assert(out0$spec==out0);
9   $assert(out1$spec==out1); ...
10 }
```

Fig. 8. Comparison transformation

```
1 #ifdef _CIVL
2 $input int nx, nst, NXB=5, NSTB=5;
3 $assume(2<=nx && nx<=NXB && 0<nst && nst<=NSTB);
4 $input double u_init[nx];
5 $output double result[nst+1][nx+2];
6 #else
7 int nx, nst; ...
8 #endif
9 void initialization() {...
10   u_prev = (double*)malloc((nxl + 2)*sizeof(double));
11   ... u_prev[0] = u_curr[0] = 0; ...
12 }
13 void exchange(){
14   MPI_Sendrecv(&u_curr[1],1, ...,&u_curr[nxl+1], ...);
15   MPI_Sendrecv(&u_curr[nxl],1, ...,&u_curr[0], ...);
16 }
17 void update() {...}
18 int main(int argc, char * argv[]) {...
19   initialization();
20   for(iter = 0; iter < nst; iter++)
21     if(nxl != 0){ exchange(); update();}
22   ...
23 }
```

Fig. 9. Excerpt of wave1d.c in MPI

```
1 Step 110: State 118, p1:
2   309->310: u_prev=...$malloc(...)
3     [u_prev:=&<d21>heap.malloc3[0][0]] at ...
4 --> State 120
5 ...
6 Step 116: State 127, p1:
7   315->316: *(&<d21>heap.malloc3[0][0]+0)=0
8     at wave1d.c:147.2-15 "u_prev[0] = 0"
9 --> State 128
10 ...
11 Step 153: State 166, p1:
12   352->159: MPI_Sendrecv(&<d21>heap.malloc4[0][1],
13     1, 23, -3, 2, ...) at ... "MPI_Sendrecv..."
14 --> State 167
15 ...
16 Step 189: State 202, p0:
17 37->38: $waitall(&<d12>_par_procs0[0], 1) at ...
18 --> State 203
```

Fig. 10. Example trace for comparing 1d-wave programs

With help of the precise information in such a trace, a developer is guided to fix *wave1d.c* by adding statements to initialize `u_next[0]` and `u_next[nxl+1]` to 0, and then CIVL is applied to confirm the fix. In fact, the erroneous *wave1d.c* had been used for two years in a parallel computing course taught by one of us (Siegel), but the defect was never noticed, apparently because the allocated memory was almost always initialized with 0s, something which is certainly not guaranteed by the C Standard.

V. Related Work and Discussion

There exist a number of verification tools for parallel programs using state exploration or symbolic execution, such as TASS [20] and ISP [21] for MPI programs, CSeq [22] and DiVinE 3.0 [23] for Pthreads programs, GKLEE [24] and SESA [25] for CUDA programs, Java PathFinder [26] and its concurrent extension jpg-concurrent [27] for Java programs, and Basset [28] for Java-based actor programs. Although these tools contribute to improving the correctness of parallel programs, they are limited to one specific parallel mechanism.

Both the modeling languages Promela of the Spin [29] model checker and CSP_M of the FDR3 [30] refinement checker support guarded statements, which is similar to CIVL's `$when` primitive. However, neither language supports dynamic constructs like procedures, pointers, and heaps, and hence are incapable of flexibly modeling various features of parallel programs. For example, both languages allow processes to be defined only in the global scope, and therefore cannot represent a hierarchical memory model like that of CUDA, which distinguishes CPU memory, global Grid memory, per-block shared memory, and per-thread local memory. In other words, most existing modeling languages are targeting high-level behaviors of computer systems, and lack the ability to model certain features of concurrency dialects. CIVL-C is proposed as a general modeling language, with features such as embedded functions, dynamic memory allocation, pointers and so on, so that complex structures of various concurrency dialects could be represented.

ThreadSanitizer [31] checks data races of C++ code dynamically and has been used for testing applications at Google. AddressSanitizer [32] detects memory access bugs, including buffer overflows and used of heap memory. CIVL also checks these features, as part of the standard properties.

SymDiff [33], [34] is a tool for the equivalence checking of two Boogie programs and could be used for various source language like C, C# for which translation to Boogie is available. It is representative of another class of equivalence-checking tools, which requires the two programs to be closely related to each other, i.e., the procedures, globals and constants of the two programs can be matched in a certain way. As for CIVL, it only requires the two programs to be compared contain the same set of input and output variables. Moreover, SymDiff lacks support for concurrency.

CIVL is not only a model checker, but also a verification framework. CIVL-C is capable of modeling a wide range of features in concurrency dialects. Further, CIVL can be easily extended to verify a new concurrency dialect by adding a front-end to translate the dialect into CIVL-C. This requires relatively little effort, due to the well-defined (and thoroughly documented) APIs provided by CIVL. For example, the MPI transformer is implemented using 572 lines of Java code and 1435 lines of CIVL-C code for implementing the library *mpi.h*; the OpenMP transformer is realized with 3536 lines of Java code and 432 lines of CIVL-C code. There is great potential for CIVL to support more concurrency dialects besides the four existing ones: MPI, CUDA, OpenMP and Pthreads.

REFERENCES

[1] Message-Passing Interface Forum, "MPI: A Message-Passing Interface standard, version 3.0," http://www.mpi-forum.org/docs/docs.html, Sep. 2012.

[2] Institute of Electrical and Electronics Engineers, Inc., *IEEE Standard for Information Technology—Portable Operating System Interface (POSIX) Base Specifications, Issue 7, IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*. 3 Park Avenue, New York, NY 10016-5997, USA: IEEE, Dec. 2008.

[3] OpenMP Architecture Review Board, "OpenMP API Specification for Parallel Programming," http://openmp.org/wp/, accessed Feb. 8, 2015.

[4] "CUDA Programming Guide Version 5.0." http://docs.nvidia.com/cuda/cuda-c-programming-guide/, accessed Feb. 8, 2015.

[5] J. Dinan, P. Balaji, E. L. Lusk, P. Sadayappan, and R. Thakur, "Hybrid Parallel Programming with MPI and Unified Parallel C," in *7th ACM International Conference on Computing Frontiers*, Bertinoro, Italy, 04/2010 2010.

[6] C. Yang, C. Huang, and C. Lin, "Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters," *Computer Physics Communications*, 2010.

[7] "ABC: ANTLR-Based C front-end," http://vsl.cis.udel.edu/abc, accessed Jul. 28, 2015.

[8] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers, "CIVL: The concurrency intermediate verification language," Nov 2015, to appear.

[9] "SARL: The Symbolic Algebra and Reasoning Library," http://vsl.cis.udel.edu/sarl, Accessed Feb. 6, 2015.

[10] C. Barrett and C. Tinelli, "CVC3," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, ser. Lecture Notes in Computer Science, W. Damm and H. Hermanns, Eds., vol. 4590. Springer, 2007, pp. 298–302.

[11] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "CVC4," in *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, ser. Lecture Notes in Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 171–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032319

[12] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963. Springer, 2008, pp. 337–340. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-78800-3_24

[13] "Lawrence Livermore National Laboratory OpenMP tutorial," https://computing.llnl.gov/tutorials/openMP/exercise.html, accessed Feb. 8, 2015.

[14] "SV-COMP 2015: Competition on software verification," http://sv-comp.sosy-lab.org/2015, Accessed Feb. 7, 2015. [Online]. Available: http://sv-comp.sosy-lab.org/2015

[15] S. F. Siegel and T. K. Zirkel, "FEVS: A Functional Equivalence Verification Suite for high performance scientific computing," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 427–435, 2011.

[16] "AMG 2013," https://codesign.llnl.gov/amg2013.php, accessed Aug. 20, 2015.

[17] "The CIVL Manual," https://vsl.cis.udel.edu/civl/test/civl-manual.pdf, Accessed Apr. 17, 2015.

[18] Center for Development of Advanced Computing, "hyPACK 2013: MPI-OpenMP Programs," http://cdac.in/index.aspx?id=ev_hpc_hypack_mpi_openmp_programs, accessed Apr. 17, 2015.

[19] S. F. Siegel and T. K. Zirkel, "A Functional Equivalence Verification Suite," http://vsl.cis.udel.edu/fevs, accessed Feb. 6, 2015.

[20] ——, "TASS: The Toolkit for Accurate Scientific Software," *Mathematics in Computer Science*, vol. 5, no. 4, pp. 395–426, 2011.

[21] G. Gopalakrishnan, R. M. Kirby, S. Siegel, R. Thakur, W. Gropp, E. Lusk, B. R. De Supinski, M. Schulz, and G. Bronevetsky, "Formal analysis of MPI-based parallel programs," *Communications of the ACM*, vol. 54, no. 12, pp. 82–91, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2043174.2043194

[22] B. Fischer, O. Inverso, and G. Parlato, "CSeq: A sequentialization tool for C," in *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'13, N. Piterman and S. A. Smolka, Eds. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 616–618. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36742-7_46

[23] J. Barnat, L. Brim, V. Havel, J. Havlícek, J. Kriho, M. Lenco, P. Rockai, V. Still, and J. Weiser, "DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs," in *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13–19, 2013. Proceedings*, ser. Lecture Notes in Computer Science, N. Sharygina and H. Veith, Eds., vol. 8044. Springer, 2013, pp. 863–868.

[24] G. Li, P. Li, G. Sawaya, G. Gopalakrishnan, I. Ghosh, and S. P. Rajan, "GKLEE: Concolic verification and test generation for GPUs," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York: ACM, 2012, pp. 215–224, http://www.cs.utah.edu/fv/GKLEE.

[25] P. Li, G. Li, and G. Gopalakrishnan, "Practical symbolic race checking of GPU programs," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC14*, Nov. 2014, pp. 179–190.

[26] C. S. Pasareanu and N. Rungta, "Symbolic PathFinder: symbolic execution of Java bytecode," in *Proceedings of ASE*, 2010, pp. 179–180.

[27] M. Ujma and N. Shafiei, "jpf-concurrent: An extension of Java PathFinder for java.util.concurrent," *CoRR*, vol. abs/1205.0042, 2012.

[28] S. Lauterburg, M. Dotta, D. Marinov, and G. Agha, "A Framework for State-Space Exploration of Java-Based Actor Programs," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 468–479.

[29] G. J. Holzmann, *The* SPIN *Model Checker*. Boston: Addison-Wesley, 2004.

[30] T. Gibson-Robinson, P. J. Armstrong, A. Boulgakov, and A. W. Roscoe, "FDR3 - A modern refinement checker for CSP," in *TACAS*, 2014, pp. 187–201.

[31] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: Data Race Detection in Practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71.

[32] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342821.2342849

[33] M. Kawaguchi, S. K. Lahiri, and H. Rebelo, "Conditional equivalence," Tech. Rep. MSR-TR-2010-119, October 2010. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=137899

[34] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo, "SymDiff: A language-agnostic semantic diff tool for imperative programs," in *Computer Aided Verification (CAV '12) (Tool description)*. Springer, July 2012. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=161804