CIVL: The Concurrency Intermediate Verification Language

Stephen F. Siegel Manchun Zheng Ziqing Luo
Timothy K. Zirkel Andre V. Marianiello John G. Edenhofner
Department of Computer and Information Sciences, University of Delaware, USA
{siegel|zmanchun|ziqing|zirkeltk|andrevm|johneden}@udel.edu

Matthew B. Dwyer Michael S. Rogers
Department of Computer Science and Engineering, University of Nebraska - Lincoln, USA
{dwyer|mrogers}@cse.unl.edu

ABSTRACT

There are many ways to express parallel programs: message-passing libraries (MPI) and multithreading/GPU language extensions such as OpenMP, Pthreads, and CUDA, are but a few. 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 among efforts.

To address this problem, we present **CIVL: the Concurrency Intermediate Verification Language**. CIVL provides a general concurrency model capable of representing programs in a variety of concurrency dialects, including those listed above. The CIVL framework currently includes front-ends for the four dialects, and a back-end verifier which uses model checking and symbolic execution to check a number of properties, including the absence of deadlocks, race conditions, assertion violations, illegal pointer dereferences and arithmetic, memory leaks, divisions by zero, and out-of-bound array indexing; it can also check that two programs are functionally equivalent.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods, model checking*; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; D.2.5 [**Software Engineering**]: Testing and Debugging—*symbolic execution*

Keywords

concurrency, verification, parallel programming, program transformation, intermediate representation, MPI, OpenMP, CUDA, Pthreads, model checking, symbolic execution

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807635

1. INTRODUCTION

There are today a number of mechanisms for expressing parallelism in computer programs. Some of the most widely-used are the message-passing library MPI [37], the multithreading library POSIX Threads (Pthreads) [27], the high-level multithreading annotation system OpenMP [38], and Nvidia's general-purpose GPU language extension CUDA [15]. New versions of these concurrency dialects, and entirely new mechanisms, appear regularly. Moreover, *hybrid* parallel programs, which combine two or more dialects, are increasingly common.

The complexity and dynamic nature of the concurrency world pose a challenge for verification researchers. Most verification techniques or tools dealing with concurrency target a single dialect. There is little exchange of code, ideas, or techniques across dialects, limiting the impact of tools and resulting in significant duplicated effort.

This paper introduces a framework (Fig. 1) to address this problem. The framework is centered around a general model of concurrency: the *Concurrency Intermediate Verification Language*. It includes a programming language, CIVL-C (Sec. 2.1), which adds to C a number of primitives dealing with concurrency and specification. The front-end, ABC [1], accepts programs written in C with any combination of the concurrency dialects listed above, as well as CIVL-C. *Transformers* replace uses of the dialects with semantically equivalent CIVL-C code, resulting in a "pure" CIVL-C program (Sec. 4). This is lowered to the CIVL intermediate representation (IR), yielding a CIVL *model* (Sec. 2.2). The idea is that new static analysis and verification techniques can be implemented at the model or AST level, and be immediately applied to programs using any of the dialects.

The framework includes a verification tool (Sec. 3), based

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.

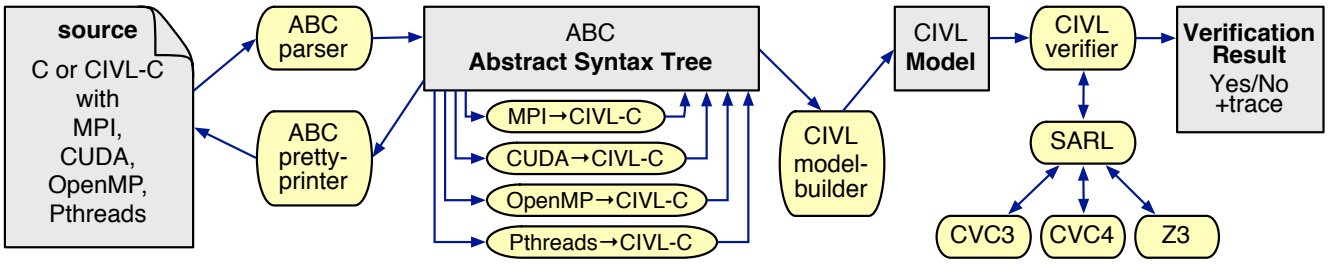

Figure 1: The CIVL framework

on model checking and symbolic execution, which can verify the following *standard properties*: absence of assertion violations, deadlocks, memory leaks, improper pointer dereferences or arithmetic, out-of-bound array indexes, reads of uninitialized variables, and divisions by 0.

In addition to the standard properties, a large number of *dialect-specific properties* are verified. For MPI, these include: for any communicator c, the sequence of collective calls made on c is the same for all processes in c; a received message fits in the receive buffer; no MPI function is called before `MPI_Init` or after `MPI_Finalize`; if `MPI_Init` is called then so is `MPI_Finalize`; and the type of data in a buffer is consistent with the `MPI_Datatype` argument used in an MPI call. For Pthreads, these include: a thread does not attempt to obtain a non-recursive mutex lock twice, terminate before releasing a non-robust mutex lock, or call `pthread_cond_wait` without holding the lock on the mutex argument. For OpenMP, accesses to shared variables that could result in undefined behavior, according to OpenMP's weak memory consistency model, are reported as errors. This is just a sample; many more such properties are checked.

Finally, the verifier can check that two programs are *functionally equivalent*, i.e., whenever the two programs are given the same input, they will produce the same output. This last feature is especially useful for verifying that a complex parallel program implemented using one or more dialects conforms to a simple sequential realization of an algorithm.

Like most model checkers, the CIVL verifier requires small concrete bounds on the numbers of processes and input sizes. It thus relies on the *small scope hypothesis*, the claim that defects in concurrent programs almost always manifest themselves in small configurations. By "manifest," we mean there *exists some execution* in the small scope which results in failure. Different executions arise from many sources: different inputs, different interleavings of statements from concurrently-executing processes, and from exploring the full range of behaviors allowed by the relevant APIs. For example, an `MPI_Send` may or may not be forced to synchronize with a matching receive, and the iteration space of an OpenMP `for` loop may be partitioned among threads in a number of ways. Hence a tool which checks that a property holds on *all possible executions* of a program within a small scope is likely to discover a violation, if one exists. (This is in stark contrast to ordinary testing, which typically explores only a very small subset of the execution space.) The user can specify these bounds on the command line, or by annotations in the original source code, or some combination of these approaches.

The verifier can also produce a *minimal counterexample* when a property violation is found. This is an execution trace of minimal length culminating in failure—something which greatly facilitates understanding, isolating, and repairing defects. Again, this contrasts with most testing and debugging methods in HPC, which often involve traces with astronomical numbers of threads, processes, or execution steps.

The language has been designed to facilitate the kinds of transformations and verification described above. Two design themes contribute to these goals: *scopes* and *processes*. While standard C allows function definitions only in file scope, CIVL-C allows such definitions in any scope; these functions can also be *spawned* to create new processes.

To illustrate how these concepts are used, consider a typical C/MPI program. Such a program contains the code for a generic process, which will be instantiated n times at runtime ($n \geq 1$). The global variables, heap, and functions in the original program become "local" to each process. There is no shared memory; communication and synchronization takes place by calls to functions in the MPI library. This program can be represented by a CIVL-C program containing one large function representing an MPI process; see Fig. 6(a). That function would essentially contain the entire original program (including its global variables and function definitions) which would be spawned n times. The global scope of this CIVL-C program would contain variables representing the message buffers, the only shared state.

CIVL is free, open source software, and can be downloaded from http://vsl.cis.udel.edu/civl. It is written entirely in Java 7 and is distributed as a JAR file. (It invokes one or more external theorem provers, which must be installed separately.) A detailed manual, API documentation, over 350 concurrency examples, and a comprehensive test suite are also included.

2. LANGUAGE

2.1 CIVL-C

CIVL-C is based on the C11 [28] dialect of C. It excludes the components of C11 dealing with concurrency, as CIVL has its own concurrency model. However, with few exceptions, any strictly conforming sequential C11 program is a legal CIVL-C program. The main restriction is that CIVL-C requires dynamically created objects to be typed, so each `malloc` call must be surrounded by a cast to a non-`void*` pointer type; this is already a standard convention in C.

The CIVL-C concurrency model is simple and flexible. Unlike C11, in CIVL-C, functions can be defined in any

`$input` : type qualifier declaring global variable to be read-only and initialized with unconstrained value of its type
`$output` : type qualifier declaring global variable to be an output, a write-only variable
`$assume(expr)` : statement informing the verifier to ignore the current execution unless *expr* holds
`$assert(expr)` : checks that *expr* holds and reports error if it does not
`$forall {T i | cond} expr` : universal quantification, i.e., $\forall i \in T.(cond \Rightarrow expr)$. `$exists` is similar
`$range` : type representing an ordered set of integers; e.g., `$range r1 = a .. b`
`$domain(n)` : type representing an ordered set of n-tuples of integers; includes *Cartesian* domains, e.g., `$domain(3) d={r1,r2,r3}`, the Cartesian product of 3 ranges in dictionary order
`$scope` : type for reference to a dyscope; includes constants `$here` (the scope in which the expression occurs) and `$root`
`$proc` : type representing reference to an executing process; includes constant `$self`
`$malloc(scope,size)` : allocates object in heap of specified dyscope; freed with `$free`
`$for (int i,j,...: d) stmt` : iterates over the tuples $\langle i, j, \ldots \rangle$ in a domain *d*
`$choose_int(n)` : expression returning an integer in $[0, n-1]$, chosen nondeterministically
`$choose { stmt1 stmt2 ... default: stmt }` : nondeterministic selection of one enabled statement
`$spawn f(arg0, ...)` : creates and returns reference p to new process executing function *f*
`$wait(p)` : waits until process p terminates then removes it from the state
`$waitall(procs, n)` : like above for n processes; *procs* has type `$proc*`
`$parfor (int i,j,...: d) stmt` : spawns processes for each element in the domain *d* and waits until all terminate
`$when(guard) stmt` : guarded command; enabled only when boolean expression *guard* evaluates to *true*
`$atomic stmt` : executes *stmt* without interleaving of other processes

Figure 2: Some commonly-used CIVL-C primitives

scope. Furthermore, these functions can be spawned to generate new processes. These basic building blocks can be combined in myriad ways to model the concurrency and memory hierarchies that arise in very different dialects. CIVL also uses a sequential memory consistency model in which every read or write to a variable occurs atomically and an execution is a simple interleaving of atomic events from the processes. In order to model more complex consistency models, one can define new primitives for accessing shared variables. We have done this for OpenMP; see Sec. 4.2.

CIVL-C adds several types and other language constructs. The most important of these are summarized in Fig. 2. The CIVL-C keywords all begin with '`$`'.

There are types for references to scopes and processes. Objects of these types can be assigned to variables, returned by functions, and passed as parameters, as with other scalar types. A `$spawn` command returns an object of `$proc` type, which can later be used as the argument to `$wait`. As mentioned in Sec. 2.2, each scope has its own heap. The function `$malloc` takes an extra argument of `$scope` type that specifies the heap in which memory should be allocated.

The `$domain` type and related functions make it easy to represent and manipulate "iteration spaces". This is a common theme. An OpenMP `for` loop nest, for example, defines a Cartesian iteration space which can be partitioned among threads in various ways. Translation of this construct uses a CIVL-C library function that takes a domain and returns a partition of it into subdomains, either nondeterministically or according to some heuristic. CUDA grids and thread blocks are indexed by integer 3-tuples which can be represented by domains. These thread groups can be launched with a single invocation of `$parfor` on the domain.

Every CIVL-C statement has an implicit *guard*, a condition that determines whether to execute the statement. For most statements the guard is *true*; an exception is `$wait`, which is enabled only when the process specified by its argument terminates. Also, a guard can be attached to any statement using `$when`. This can be used to program low-level concurrency constructs such as semaphores. Mechanisms for nondeterministic choice are provided by `$choose` and `$choose_int`.

Several primitives deal with specification. There are statements to `$assume` and `$assert` predicates, and first-order quantifiers `$forall` and `$exists`. The `$input` and `$output` qualifiers facilitate specification of functional equivalence of two programs (Sec. 3).

A number of additional functions and abstract datatypes are provided in the CIVL library. These are used to model aspects of the concurrency runtimes that would be too tedious and inefficient if modeled in ordinary C—and too far removed from the logical theories supported by theorem provers. Examples include a function for determining "deep equality" of any two objects; a `$bundle` type along with a function to pack any contiguous region of memory into a bundle and another to unpack a bundle into a specified region; a *sequence* type supporting insert, delete, and append operations; a *barrier* object with functions for creating, joining, invoking, and destroying barriers; and a *communicator* type `$comm` comprising a set of FIFO channels with functions to insert, remove, and query messages.

2.2 Semantics

A CIVL-C AST is transformed to a *CIVL model*, a lower-level representation with a precise, mathematical semantics. We briefly sketch some of the main semantic concepts of CIVL models; for full details, see the CIVL Manual.

Each model specifies some set Σ of *static scopes*. This has the structure of a rooted tree. The elements of Σ correspond to the lexical scopes in the program: σ is a child of σ' if σ is contained immediately (with no intervening scope) in σ'. Fig. 3 (left) shows a CIVL-C program, with the static scopes numbered. Fig. 3 (middle) shows the corresponding static scope tree. Fig. 3 (right) shows a state reached during verification of the example program.

A model associates to each $\sigma \in \Sigma$ a set of variables $\mathsf{vars}(\sigma)$: the variables declared in σ. For each σ, $\mathsf{vars}(\sigma)$ includes a *heap variable* which is special in that it can be modified only through the system functions `$malloc` and `$free` (Sec. 2.1).

A model specifies a set of *function symbols*, which in-

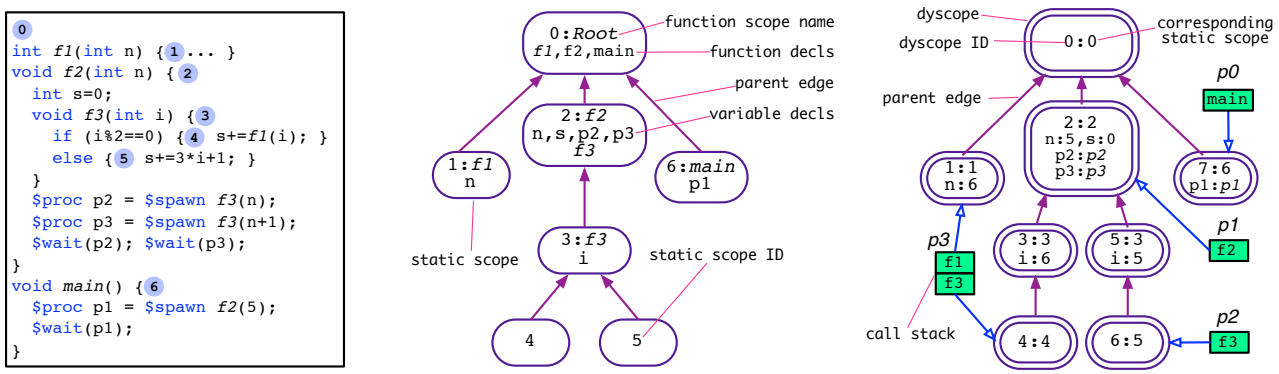

Figure 3: A CIVL-C program with static scopes numbered; its static scope tree; and a state.

cludes a *root function*. The function bodies themselves are represented by program graphs: directed graphs in which nodes correspond to locations and transitions represent an atomic execution step. Each transition comes with a guard (a boolean expression which specifies when that transition is enabled) and a primitive statement which specifies variable updates. Some functions f are *system functions*: instead of a program graph, the behavior of f is given by a function which specifies how the variables in a state are updated when f is called. The update happens in a single atomic step. In CIVL-C, such functions come with a Java class which performs the update.

A *state* of a model M consists of a set Δ of *dynamic scopes* (or *dyscopes*), which also has the structure of a rooted tree, together with a tree homomorphism $\mathsf{static}\colon \Delta \to \Sigma$. If $\mathsf{static}(\delta) = \sigma$, we say δ is an *instance* of σ. For each $\delta \in \Delta$, the state specifies a value for each variable in $\mathsf{vars}(\mathsf{static}(\delta))$.

The state also specifies a set of processes. Each process p has a *call stack*, which is a finite sequence of *frames*. Each frame specifies a location in a program graph and a dyscope which forms the evaluation context. We say p *reaches* dyscope δ in state s if there exists a path from a dyscope occurring in p's call stack to δ, following the parent edges in Δ. In the state of Fig. 3 (right), for example, $p3$ reaches five dyscopes, with IDs $0, \ldots, 4$. A dyscope δ is *reachable* in s if there is some process which reaches δ in s.

Execution follows the standard interleaving model beginning in an initial state with one process with a single frame whose values are "undefined". The guards of the transitions emanating from the locations occurring at the tops of the call stacks are evaluated; one of the transitions with guard evaluating to *true* is chosen, its statement executed, and the state is updated. For the most part, this is standard, but a few points are special. First, whenever a process moves from a static scope σ to a new scope σ', a sequence of new dyscopes is added corresponding to the chain from the lowest common ancestor of σ and σ' to σ'. A *call* pushes a new frame on the call stack and moves control to the scope associated to the start location of the called function, which entails the creation of new dyscopes as just described. A *spawn* is similar, but creates a new process and pushes the frame onto its call stack. A *wait* is enabled only when the process being waited on terminates; executing the *wait* removes the terminated process from the state. If a dyscope becomes unreachable, it is also removed from the state.

3. VERIFICATION

Commands.

The CIVL verifier is invoked on the command line by `civl verify` *`[options] filenames`*. This marshals together all of the tools in the framework to (1) preprocess and parse each file, (2) merge the resulting translation units into a single AST representing a whole program, (3) deploy the appropriate transformers, as determined by the headers and constructs used in the program, to yield a pure CIVL-C AST, (4) build a CIVL model from the AST, and (5) run the CIVL verifier to verify or refute the standard properties.

Depending on the value of option `-errorBound`, the verifier may stop after discovering the first violation, or continue searching for more. In any case, the violations are categorized and logged. The categories include *assertion violation*, *deadlock*, *memory leak*, and so on. A description of the violation and a compact representation of the trace leading to it are included in the log. Two violations are considered equivalent if they have the same type and involve the same code location(s). The log keeps only a single representative from each equivalence class—the one with shortest trace. At the end of the search, a detailed report of the results is saved to disk, and a summary is printed to the terminal.

The command `civl replay` plays back a trace recorded in the log. Depending on the options, it may print every transition and/or state along the trace. The transitions show the statement being executed and all variable updates resulting from its execution. The state shows the values of all variables in every dyscope, and the call stack of every process. When reporting violations and displaying traces, CIVL provides references to the original source, giving file names, line and column numbers, and an excerpt of the surrounding text.

The command `civl show` displays any combination of the following: the results of preprocessing and parsing, the original AST, the AST resulting after each transformation, and the final CIVL model. The ASTs may be printed either in a hierarchical plain-text format, or as CIVL-C code.

The command `civl compare` verifies the functional equivalence of two programs. The first program is considered the *specification* and the second the *implementation*. For each `$input` variable in the specification there must be a corresponding `$input` variable (with same name and type) in the implementation. The two programs are combined

into a single *composite* program and then verified. In the composite, the original programs are enclosed in two separate functions. The `$input` and `$output` variables are pulled into the root scope, but whereas the inputs are unified, each function writes to its own distinct output variables. The composite's *main* function invokes the two functions in sequence and then asserts that the corresponding pairs of output variables agree. If those assertions cannot be violated, the specification and implementation must produce the same output whenever given the same input—they are functionally equivalent. This transformation is compatible with the others, so that an MPI+Pthreads program can be compared with a sequential one, for example.

The command `civl run` executes the program by resolving all nondeterministic choices randomly. A random seed can be specified for reproducibility. Finally, `civl help` summarizes all commands and options.

Symbolic execution.

The general approach taken by the verifier, symbolic execution, is well known; see, for example, [29, 30, 50]. The basic idea is to explicitly enumerate the reachable states of a CIVL model, but using symbolic expressions instead of concrete values for variable values. The state also includes a *path condition* variable `pc`, which holds a symbolic expression of boolean type. Initially *true*, `pc` is updated when executing a transition with a nontrivial guard g: the new value of `pc` is the conjunction of the old value of `pc` and the result of evaluating g. Hence `pc` records the history of the (branch and other) choices made along the current path. At any point, if `pc` is determined to be unsatisfiable, the current path is infeasible (does not correspond to any concrete execution) and the search backtracks.

CIVL uses the *Symbolic Algebra and Reasoning Library* (SARL) [48] to create, manipulate, and simplify symbolic expressions, and to determine the validity (or dually, the satisfiability) of first-order formulas involving those expressions. SARL essentially combines the services of a symbolic algebra tool such as Mathematica and those of an SMT theorem prover. SARL is particularly effective at simplifying expressions involving multivariate polynomials, including quotients of such polynomials, and can resolve many validity queries through the simplification process alone. For those that it cannot resolve itself, it invokes one or more in a series of automated theorem provers until a conclusive result is obtained. For the experiments in this paper, SARL used CVC4 [6], Z3 [16], and CVC3 [7]. CIVL uses SARL's ideal (mathematical) models of integers and reals; this is generally preferred for equivalence-checking, since a parallel numerical program is rarely expected to be "bit-level" equivalent to its sequential specification.

Internally, pointer values are represented as tuples comprising (1) a reference to a dyscope, (2) a reference to a variable within that scope, and (3) a sequence of "navigators" to specify a sub-component of an object, e.g., field 3 of element 5 of an array of structs. Hence the verifier uses a logical, not physical, model of memory. Nevertheless, it is adept at performing most kinds of pointer arithmetic that have defined behavior according to the C11 Standard; those that result in undefined behavior, such as pointers beyond the bounds of an object, are reported as errors.

States are represented exactly as described in Sec. 2.2 and depicted in Fig. 3. They are also immutable, facilitating sharing of common sub-components, such as dyscopes and call stacks, among distinct states. Immutability requires the generation of new states for each transition, but only a small portion of the state is changed for any transition and the rest is shared, by copying references, which makes state generation very efficient in space and time. Since CIVL states are typically large, this is essential in reducing the memory footprint. CIVL incorporates well-known approaches to managing states, e.g., "canonicalization" and "concretization" of symbolic values constrained to a singleton set.

The analysis performed by CIVL is *conservative*. This means that if the verifier returns the result "all properties hold" then all properties hold on all executions of the program (of course, within the specified bounds). However if CIVL reports a violation, it is possible for that violation to be *spurious* (a "false alarm"). Spurious reports arise because there are validity queries for which SARL and the underlying provers return an inconclusive result ("unknown"). In such cases the user may manually inspect the resulting trace and/or insert assumptions into the code which eliminate the spurious result.

CIVL also prioritizes the violations it finds by their *certainty*. The highest level of certainty, `CONCRETE`, means that concrete values have been determined for all inputs which satisfy the path condition and cause the assertion to evaluate to *false*; next is `PROVABLE`: a theorem prover has declared the path condition to be satisfiable and the assertion to be invalid, but has not produced concrete witnesses for these facts; `MAYBE` indicates all provers have returned inconclusive results on one or both of these questions; `UNKNOWN` indicates some situation that CIVL cannot handle and no prover invocation is involved. The log orders the violations by decreasing certainty.

Partial Order Reduction.

Partial order reduction is an essential optimization for model checking [23]. Given a state s, the goal is to find a small set of processes P such that only the transitions from P need to be explored from s, while still guaranteeing that if a property violation exists, one will be found. Generally, one searches for a set P satisfying the following: on any execution departing from s, no transition *dependent* on a transition in P can occur without a transition in P occurring first. The set of enabled transitions in P is known as an *ample set*. The standard example is a two-process program with a state in which each process is about to modify some process-local variable; in this case P can be taken to be a singleton set containing one of the two processes.

The situation with CIVL is more complicated, since there is a hierarchy of scopes which can be shared by multiple processes at different points, and each dyscope may contain a heap. To determine if some candidate set of processes P can be used to form an ample set, it is necessary to first consider all the dyscopes these processes can reach. From the non-heap variables in those dyscopes, one follows the pointer edges to determine all objects that can be reached by pointer dereferencing. The result is some set S of reachable objects. If no process outside of P can reach any of the objects in S, then P can form an ample set: no process outside of P can reach an S object unless a process in P executes first. (This strategy generalizes that of [17].)

Consider the example in Fig. 4. In this state, a dashed arrow indicates that an object contains a pointer into another

object. Process p_0 has three visible variables, `x`, `q`, and `p`, and reaches $\{o_3, \ldots, o_8\}$; p_1 reaches every object except o_2 (which is unreachable, and represents a memory leak); p_2 reaches $\{o_3, o_7\}$. Process p_0 is at a location with exactly one outgoing statement, `x++`. It follows that p_0 alone cannot form an ample set, since this statement accesses o_6, and p_1 reaches o_6. On the other hand, $\{p_0, p_1\}$ forms an ample set, since p_2 reaches neither o_6 nor o_8. Finally, $\{p_2\}$ does not form an ample set, since p_0 (or p_1) reaches o_7. Note how the organization of the state into scopes enables a precise representation of the parts of the state that a process can "reach", which is key to making this analysis precise.

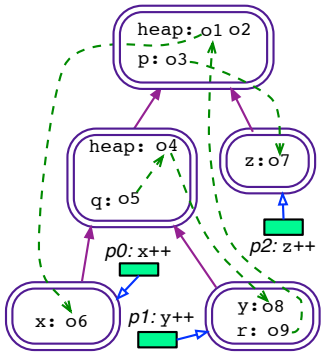

Figure 4: POR

The CIVL verifier searches for an ample set by iterating over all processes p. For each p, the reachability analysis described above is used to generate a set of processes including p that forms an ample set. From these, a set with the minimum number of processes is selected.

Specification.

A preprocessor macro `_CIVL` is defined when using the CIVL verifier. By using `#ifdef` directives, one can embed CIVL-C code which is used only when verifying the program. In many of the complex numerical examples, we exploit this mechanism to include `$input` variables used to initialize program data structures in verification mode only. We have also used this to insert simple sequential code to compute an oracle that is compared with the results of the parallel program; this is a "one-file" alternative to using `civl compare` and is equally effective.

Limitations.

In addition to the restriction on the use of `malloc` (Sec. 2.1), the verifier treats C's bit-wise operations as uninterpreted functions. This means that if the correctness of a program depends on some subtle semantics of those operations, CIVL will likely report spurious errors. Also, most, but not all, of the standard C library is supported.

Each transformer currently accepts a significant subset—but not all—of its source dialect. For MPI, standard mode blocking point-to-point operations and all collective operations are supported; support for nonblocking operations and more advanced MPI features is in progress. Pthreads support includes thread creation, termination, waiting, attribute objects, and synchronization through barriers, mutexes, conditions, spin locks, and read-write locks; but not cancellation, detachment, realtime threads, `pthread_key_t`, or `pthread_once`. The CUDA transformer requires a single device; graphics-specific capabilities (e.g., textures) are not supported. The OpenMP transformer supports parallel, worksharing, and master and synchronization constructs, along with data sharing clauses, but not yet `simd`, `task`, `device` and `cancel` constructs. As discussed in Sec. 4, the OpenMP transformer models OpenMP's weak memory consistency model in full fidelity, but the Pthreads and CUDA transformers currently assume sequential consistency.

4. ANALYSIS AND TRANSFORMATION

Programs using the concurrency dialects are transformed into "pure" CIVL-C programs through a combination of three techniques: (1) high-level restructuring, involving the creation of new functions and scopes and the re-organization of code; (2) localized replacement of targeted constructs with equivalent CIVL-C code, and (3) custom implementations of concurrency library functions in CIVL-C. For dialects that are purely library-based, such as Pthreads and MPI, most of the work falls under (3), though some of (1) is also required. For OpenMP, a pragma-based dialect, and CUDA, a language extension and library, the emphasis is much more on (2), though (1) and (3) also come into play.

As illustrated in Fig. 1, our framework allows multiple transformations to be applied to a single program. This has the advantage of controlling the complexity of the transformations and enabling the composition of multiple transformations—to support the analysis of hybrid programs.

Name	ABC	Trans.	Lib.
OpenMP	975	3536	432
OpenMP Simplify	0	738	0
Pthreads	0	603	987
CUDA	88	563	537
MPI	0	572	1435

Figure 5: Number of non-comment source lines of dialect-specific code in ABC, Transformer, and Library

Transformations are designed as AST rewrite rules that are applied during a series of traversals of the CIVL-C AST. Rules are triggered by matching specific AST structures such as `#pragma omp` or a call to a function named `pthread_exit` or `MPI_Init`, and they modify the matched AST structure—and related structures—as appropriate. Rules are designed to be non-interfering so that sequences of transformations can be applied.

The architecture of the CIVL framework allows for concurrency dialects to be supported at modest cost—1000 to 4000 lines of code, as shown in Fig. 5. Support for a dialect is spread across three components: ABC grammar and language processing extensions, a custom Transformer, and custom library support. This has allowed five developers to build support for OpenMP, Pthreads, CUDA, and MPI; none of those developers are the primary developers of CIVL and two are undergraduates. Fig. 6 shows sample transformations for several concurrency dialects.

For each dialect, we have defined a support library, written in CIVL-C. That library defines types, constants, and functions which are used in the transformed code. All primitives in the OpenMP support library have names beginning with `$omp_`. For MPI, CUDA, and Pthreads, the prefixes are `$mpi_`, `$cuda_`, and `$pthread_`, respectively.

The functions defined in the support libraries use many of the general-purpose CIVL-C primitives mentioned at the end of Sec. 2.1, including those dealing with sequences, barriers, FIFO channels, and domains. The support libaries are in turn used to implement the official libraries specified by each dialect: `mpi.h`, `pthread.h`, etc. Finally, a transformer may introduce functions or variables which are defined in the transformed code itself (and not part of a library). The names of these constructs begin with `_mpi_`, `_pthread_`, etc.

Space does not permit a complete description of the libraries here, but the details can be understood by reading their source code, which is part of the CIVL distribution.

4.1 Controlling access to shared state

In any concurrency dialect there is some notion of shared state: in MPI this contains the buffered messages; in OpenMP or Pthreads, the shared variables. The prevalence of statements that access that state—send and receive operations in message-passing, reads and writes in threading libraries—may lead to a combinatorial explosion in analysis techniques such as model checking. We have already seen that the CIVL verifier attempts to limit this damage using a general POR algorithm, but even this is not as good as it could be in specific situations.

As an example, consider a shared object used to store the buffered messages in an MPI program with n processes. In CIVL, this structure contains a set of n^2 FIFO queues, one for each ordered pair of processes. This object is accessed only in very specific ways: by enqueuing or dequeuing in the appropriate queue. The generic POR algorithm must assume that any two operations on this structure could interfere with each other. However, it is clear that they cannot interfere if they access different queues or if one operation is a send and the other a receive.

The CIVL framework provides a way for library developers to encode special knowledge about independence of library operations. A library can completely control a type in the following sense: the only way to create objects of that type are by a call to a function in that library, which returns to the user an opaque handle to the new object. Code outside of the library can only access the object through such a handle, making it possible to know that certain library calls must commute, regardless of what happens outside of the library. Library calls can accept a scope parameter which they pass to `$malloc` to control where they allocate memory, e.g., `$mpi_gcomm_create`, `$pthread_gpool_create`, and `$omp_gteam_create` in Fig. 6. The dialect library developer can encode this information by implementing a certain Java interface called an `Enabler`.

In the MPI case, there is such a data structure called a *global communicator*. This is located in the heap of the shared scope, and wraps together the message queues as well as meta-data on the state of processes which have joined the communicator. The handle to this object, which has type `$mpi_gcomm` in line 5 of Fig. 6(a), is visible to all processes. Each process also has within its local scope a *local communicator* object of type `MPI_Comm`, line 8. This is also a handle object which is basically an ordered pair consisting of a handle to the global object and a PID.

The library "knows" that a send or receive operation (excluding "wildcard receives") using a local communicator handle can never be impacted by another process that cannot reach that handle. If the process making the call is the only one that can reach its local handle, that single process can form an ample set. In a hybrid MPI-Pthreads program, two or more threads in one MPI process may reach that process' local communicator handle, but not threads in another process. In this case the verifier will deduce automatically that the threads of one process can form an ample set, but a single thread can not.

This pattern is used repeatedly. The Pthreads transformation provides a global and thread-local handle for accessing the thread pool; see lines 6-7 and 19-20 in Fig. 6(c). OpenMP thread teams and shared variables are treated similarly; see lines 2-5, 7-8, and 10-12 in Fig. 6(d). In each case, a small `Enabler` class is provided.

As illustrated in Fig. 6(b), there is a very direct mapping from the nested structure of the CUDA threading hierarchy (grids composed of thread blocks composed of threads) to nested CIVL functions executed in parallel using `$parfor`; see line 4 of Fig. 6(b). Function nesting, as on lines 6-12 of Fig. 6(b), limits visibility and enhances POR effectiveness.

4.2 Replacement of concurrency constructs

Multi-threading in OpenMP is achieved primarily through the use of the `omp parallel` pragma which defines an execution context that implicitly forks and joins a set of threads. Worksharing constructs include `omp for`, which defines parallel loop execution, `omp sections`, which defines a set of separate code regions that execute in parallel, and `omp single`, which defines a region to be executed by a single thread. Non-trivial OpenMP programs generally also use synchronization primitives, such as `critical` and `barrier`, and mechanisms to control data sharing, such as `private` and `shared`.

The OpenMP transformation consists of about 3500 lines of code that serve to expand the implicit semantics of OpenMP primitives. A source of complexity in those semantics is the weak consistency memory model which requires explicit management of local and global memory views and flush operations to make them consistent.

Fig. 6(d) illustrates this complexity on a small OpenMP fragment which initializes an array. The corresponding CIVL-C code manages sets of threads, grouped into teams, uses the `$parfor` construct to iteratively fork and join a set of threads, and for each `shared` variable creates a set of variables that provide the data views necessary to realize OpenMP's weak-memory model.

More specifically, `_omp_a_local` provides a thread-local view of `a`'s values. Per-thread views are coordinated through a per-team shared variable, `_omp_a_shared`, which is in turn coordinated through a variable shared by all thread teams, `_omp_a_gshared`. View coordination is achieved through calls to `$omp_write`, e.g., lines 18-19, which makes local and team shared views consistent, and `$omp_barrier_and_flush`, line 21, which makes team and global shared views consistent. These calls access `_omp_a_status` which records the threads that have accessed the variable since the last flush. Tracking this meta-data permits CIVL to detect when shared variable accesses exhibit "undefined behavior" as defined by the OpenMP Standard.

OpenMP's memory model and the semantics of thread scheduling for parallel loops create significant challenges for efficient verification. According to the OpenMP specification, an `omp for` loop with n iterations and a team of k threads gives rise to k^n schedules. Iteration domain abstractions, constructed on lines 14-15, ensure that all of those possible loop schedules are explored.

Fortunately, many OpenMP programs are written so that it is possible to determine, via static analysis, that parallel loop iterations and code sections are independent. Independence allows an OpenMP program to be *sequentialized* and can lead to significant reductions in verification.

The CIVL toolset contains an OpenMP simplifier that targets array-based parallel loops. It implements a conserva-

```
⟨external-definitions⟩
int main(void) { ... }
```

original MPI code

transformed to CIVL-C

```
$input int _mpi_nprocs;
$input int _mpi_nprocs_lo, _mpi_nprocs_hi;
$assume(_mpi_nprocs_lo <= _mpi_nprocs &&
    _mpi_nprocs <= _mpi_nprocs_hi);
$mpi_gcomm _mpi_gcomm =
    $mpi_gcomm_create($here, _mpi_nprocs);
void _mpi_process(int _mpi_rank) {
  MPI_Comm MPI_COMM_WORLD =
      $mpi_comm_create($here, _mpi_gcomm, _mpi_rank);
  ⟨external-definitions⟩
  int _gen_main(void) { ... }
  _gen_main();
  $mpi_comm_destroy(MPI_COMM_WORLD);
}
void main() {
  $parfor (int i : 0 .. _mpi_nprocs-1)
    _mpi_process(i);
  $mpi_gcomm_destroy(_mpi_gcomm);
}
```

(a) MPI

```
__global__ void add(int *a, int *b, int *c) {
  int tid = blockIdx.x;
  if (tid < N) c[tid] = a[tid]+b[tid];
}
int main(void) {
  ... add<<<gridDim,blockDim,0,stream>>>(a, b, c); ...
}
```

original CUDA code

transformed to CIVL-C

```
void $cuda_run_procs(dim3 dim, void process(uint3)) {
  $domain(3) dom = ($domain){0 .. dim.x-1,
    0 .. dim.y-1, 0 .. dim.z-1};
  $parfor(int x,y,z : dom) process((uint3){x, y, z});
}
void _cuda_add(dim3 gridDim, dim3 blockDim,
    size_t _cuda_mem_size, cudaStream_t _cuda_stream,
    int *a, int *b, int *c) {
  void _cuda_kernel($cuda_kernel_instance_t
    *_cuda_this, cudaEvent_t _cuda_event) {
    void _cuda_block(uint3 blockIdx) {
      void _cuda_thread(uint3 threadIdx) {
        int tid = blockIdx.x;
        if (tid<N) c[tid]=a[tid]+b[tid];
      }
      $cuda_run_procs(blockDim, _cuda_thread);
    }
    $cuda_wait_in_queue(_cuda_this, _cuda_event);
    $cuda_run_procs(gridDim, _cuda_block);
    $cuda_kernel_finish(_cuda_this);
  }
  $cuda_enqueue_kernel(_cuda_stream, _cuda_kernel);
}
int _gen_main(void) {
  ... _cuda_add(blocksPerGrid, threadsPerBlock, 0,
    stream, a, b, c); ...
}
int main(void) {
  $cuda_init();
  _gen_main();
  $cuda_finalize();
}
```

(b) CUDA

```
⟨external-definitions⟩
void* run(void *arg) { ... }
int main(void) {... pthread_create(...,run,...); ...}
```

original MPI+Pthreads code

transformed to CIVL-C

```
...
void _mpi_process(int _mpi_rank) {
  MPI_Comm MPI_COMM_WORLD =
      $mpi_comm_create($here, _mpi_gcomm, _mpi_rank);
  // Pthread library definitions ...
  $pthread_gpool_t $pthread_gpool =
      $pthread_gpool_create($here);
  int pthread_create(pthread_t *thread, ...,
      void *(*start)(void*), void *arg) {
    $atomic{
      thread->thr = $spawn start(arg); ...
      $pthread_gpool_add($pthread_gpool, thread);
    }
    return 0;
  }
  // ...  more Pthread library definitions ...
  ⟨external-definitions⟩
  void* run(void *arg) {
    $pthread_pool_t _pthread_pool =
        $pthread_pool_create($here, $pthread_gpool);
    ...
    $pthread_exit((void*)0, _pthread_pool);
  }
  int _gen_main(void) {
    ... pthread_create(..., run, ...); ...
    $pthread_exit_main((void*)0);
  }
  ... _gen_main(); ...
}
```

(c) MPI+Pthreads

```
#pragma omp parallel for shared(a)
for (int i=0; i<N; i++) a[i] = 0.0;
```

original OpenMP code

transformed to CIVL-C

```
int _omp_nthtreads = 1 + $choose_int(_omp_num_threads);
$omp_gteam _omp_gteam =
    $omp_gteam_create($here, _omp_nthreads);
$omp_gshared _omp_a_gshared =
    $omp_gshared_create(_omp_gteam, &a);
$parfor (int _omp_tid :  0 .. _omp_nthreads-1) {
  $omp_team _omp_team =
      $omp_team_create($here, _omp_gteam, _omp_tid);
  double _omp_a_local[N]; int _omp_a_status[N];
  $omp_shared _omp_a_shared =
      $omp_shared_create(_omp_team, _omp_a_gshared,
      &_omp_a_local, &_omp_a_status);
  $domain(1) _omp_loop_dom = ($domain){0 .. N-1 # 1};
  $domain(1) _omp_iters =
      $omp_arrive_loop(_omp_team, 0, _omp_loop_dom, 2);
  $for (int i : _omp_iters) {
     double _omp_write0 = 0.0;
     $omp_write(_omp_a_shared, &_omp_a_local[i],
         &_omp_write0);
  }
  $omp_barrier_and_flush(_omp_team);
  $omp_shared_destroy(_omp_a_shared);
  $omp_team_destroy(_omp_team);
}
$omp_gshared_destroy(_omp_a_gshared);
$omp_gteam_destroy(_omp_gteam);
```

(d) OpenMP

Figure 6: Transformation examples (lightly edited for clarity)

tive array-dependence analysis by exploiting OpenMP data sharing declarations and semantics to formulate constraints whose satisfiability assure the absence of loop-carried dependences much like analyses in the literature, e.g., [42]. The simplifier uses SARL (Sec. 3) to solve constraints.

In many cases, `omp for` and enclosing `omp parallel` constructs can be completely removed based on the thread independence; the simplifier would transform Fig. 6(d) into

```
1 for (int i=0; i<N; i++) a[i] = 0.0;
```

Partial simplification of OpenMP constructs can be performed. For instance, when not all `omp for` within an `omp parallel` are independent, those that are can be replaced with `omp single` constructs. An `omp single` requires that k schedules be explored—a significant reduction from k^n.

5. RELATED WORK

A problem similar to the one discussed in this paper arises in the verification of *sequential* programs: designing a common verification framework for a wide variety of programming languages. Recently, there have been significant advances addressing this problem. For example, the SMACK verifier [45] uses Boogie [34] as a common intermediate verification language and translates from the LLVM IR to Boogie; the result is a verification framework that can be applied to any language for which there is an LLVM front-end.

We considered using Boogie, but Boogie (and SMACK) are oriented more towards deductive verification; CIVL is geared more towards model checking and symbolic execution. For verifying concurrent programs, the second approach is much more mature and proven. Moreover Boogie does not provide a way to define functions in nested scopes, which is so essential to the CIVL concurrency model. We considered working from LLVM, but decided that language is too "low-level", for example, losing important type and scope information, and the bounds on `for` loops, all of which CIVL uses extensively.

Many verification tools use state exploration or symbolic execution for a specific concurrency dialect. Examples include TASS and ISP [24] (C/MPI), Java PathFinder [40] (Java), CSeq [19] and DiVinE 3.0 [5] (C/Pthreads), GKLEE [35] and SESA [36] (C/CUDA). The structure of the CIVL state is inspired by earlier work on Chapel verification [55]. CIVL's guarded command language is similar to Promela, the language of the model checker Spin [26]. However, Promela lacks many dynamic constructs such as procedures, pointers, and heaps, and processes can be defined only in the global scope.

The IR used by the Bandera model checking platform used a similar guarded transition system representation and supported an extensible type system [13]. Zing [2,54] 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 [34] and Why3 [18] 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.

Some CIVL features are of course found in programming languages. CIVL's `$range` and `$domain` types are borrowed from Chapel [11]. While C does not allow nested procedures, the GCC extension of C does [51, §6.4]. UNIX introduced the C `fork` and `wait` procedures [46], 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 [21] and Erlang [3] also provide `spawn` primitives, but don't allow nested procedures. One language which does provide all these features is Racket [20], and in fact the basic structure of a CIVL model can be represented in Racket in a straightforward way, though it is not clear how easily many other aspects of C (e.g., pointers and heaps) could be represented.

Other recent language designs targeting parallel execution have adopted the notions of *regions*. Phalanx [22] uses a "place" hierarchy, which is similar in some respects to CIVL scopes, and a form of region-based pointer categorization. ParaSail [39] 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.

6. EVALUATION AND DISCUSSION

CIVL is intended to provide broad support for C programs written in modern concurrency dialects. In this section, we present data from an evaluation that focuses on demonstrating the diversity of concurrency dialects and constructs that CIVL supports.

6.1 Evaluation Results

We gathered a set of C programs written in a variety of concurrency dialects, from a variety of sources, with the goal of covering a large subset of the constructs appearing in dialects. We chose examples that were offered by their user communities, e.g., the LLNL OpenMP online tutorial exercises [32], or that had been used in previous analysis efforts, e.g., the SV-COMP Pthreads concurrency benchmarks [52]. The 34 programs reported on explicitly in this section comprise 3741 non-comment source lines of code (SLOC), but they represent a small fraction of the 366 programs and 27k SLOC of code analyzed by CIVL http://vsl.cis.udel.edu/civl/sc15. We are working on multiple large case studies (10s of thousands of SLOC), but were not able to find open source applications of moderate size that used the dialects that we targeted.

With few exceptions, the only modifications performed to these programs were to support command line parameterization of quantities that determine the problem *scale*, e.g., matrix size (`NROWS`, `NCOLS`), number of time steps in simulations (`NSTEPS`), numbers of threads (`NT`) or processes (`NP`).

We added non-trivial assertions to those examples which did not already contain them. For those that perform a numerical computation, we also checked that the intermediate and/or final results agree with those of a simple sequential version, using the techniques described in Sec. 3. And of course all of the standard and dialect-specific properties (Sec. 1) were checked in every case.

Fig. 7 presents data on 34 representative examples from our evaluation. The "Type" column indicates the concurrency dialect(s) used: C=CUDA, M=MPI, O=OpenMP, P=Pthreads, and two-letter codes indicate hybrids. For OpenMP the default configuration explores the full space schedules and applies the simplifier, which targets `omp for` constructs. A superscript indicates that the simplifier is

Type	Example	R	LoC	States	Transitions	Time	Mem	ValidCalls	Prove	Scale
O	dotProduct1.c [32]	+	24	124	123	2	915	32	3	1≤NT≤3,N=8
O	dotProduct_critical.c [41]	-	35	23656	23656	6	916	28248	7	1≤NT≤3,N=100
O	matProduct1.c [32]	+	66	651	650	2	915	471	3	1≤NT≤3,NRA=8,NCA=8,NCB=8
O	heated_plate_openmp.c [44]	+	160	228034	229308	33	1126	472742	7	1≤NT≤3,M=5,N=5,EPSILON=0.1
O	fig3.10-mxv-omp.c [12]	+	62	821	820	2	915	2047	4	1≤NT≤3,M=10,N=10
O	quad_openmp.c [8]	+	91	8138	8137	13	1398	2035	5	1≤NT≤3,N=100
O	pi.c [41]	+	70	44004	44088	10	1367	54705	10	1≤NT≤3,N=100
O^r	heated_plate_openmp.c [44]	+	160	1684918	1700505	103	607	1135138	8	1≤NT≤3,M=5,N=5,EPSILON=0.1
O	omp_bug5.c [32]	-	54	2647	2652	4	916	3035	7	1≤NT≤3,N=10
M	diffusion1d.c [49]	+	164	118272	117440	30	1120	463080	146	1≤NX,NSTEPS≤5,1≤NP≤3
M	diffusion2d.c [49]	+	274	489379	485418	247	1859	2473368	49	1≤NX,NY,NSTEPS≤5,NPX=NPY=2
M	mpi_prime.c [31]	+	105	28281	28276	14	1385	79342	382	{PRIMES}⊆[10, 15],1≤NP≤4
M	mpi_pi_send.c [31]	+	120	112922	112357	101	732	305872	4241	1≤DARTS,ROUNDS,NP≤2
M	sum_array.c [49]	+	72	81852	81366	13	1393	439555	89	1≤NX≤20,1≤NP≤5
M	wave1d.c [49]	+	194	98091	97216	54	897	420943	240	1≤NX,NSTEPS≤5,1≤NP ≤3
M	wave1dBad.c [49]	-	192	496	495	4	650	2118	46	1≤NX,NSTEPS≤5,1≤NP≤3
M	gausselim.c [49]	+	293	408185	406073	115	628	1769614	1911	1≤NROW≤4,1≤NCOL≤2,1≤NP≤3
M	matmat_mw.c [49]	+	104	85982	85294	18	1315	646793	36	1≤M,N,L≤3,1≤NP≤4
C	cuda-omp.cu [53]	+	99	9401	10331	8	1409	142221	92	1≤NBLK≤4,1≤NTperBLK≤2
C	dot.cu [47]	+	99	13713	13921	6	650	110745	73	1≤N≤6, 1≤NTperBLK≤4
C	mm.cu [43]	-	146	877	878	3	515	3632	15	NBLK=4,NTperBLK=1
C	sum.cu [25]	+	72	1297	1314	3	515	15679	6	NBLK=4,NTperBLK=2
C	vectorAdd.cu [14]	+	148	4796	5179	5	650	69055	15	1≤N≤6, 1≤NTperBLK≤4
P	bug4.c [33]	-	85	12162	12597	4	650	37915	3	NT=3, ITR=5, THD=7, NSTEPS=10
P	queue_ok_longest_...c [52]	+	126	68364	71574	16	843	121365	2	SIZE=800
P	read_write_lock_...c [52]	-	38	1758	1878	2	515	1762	0	NT=4
P	sync01_true...c [52]	+	42	320	329	1	515	602	0	NT=2
P	03_incdec_true...c [52]	+	56	448	453	1	515	116	3	NT=3
MO	pie-calculation.c [9]	+	65	622	619	6	1387	1350	6	NP=2,1≤NT≤3,INTERVALS=6
MO	pie-calculation.c [9]	+	66	6590	6587	7	1390	5338	6	NP=2,1≤NT≤10,INTERVALS=100
MO^a	pie-calculation.c [9]	+	65	5326595	5350928	1412	2157	14625283	10	NP=2,1≤NT≤3,INTERVALS=6
MP	mpithreads_both.c [33]	+	87	32908	35778	10	1384	92395	5	NP=2, NT=2, VLEN=5
MP	MP-infinity-norm.c [10]	+	146	2861	2896	6	650	6228	39	NP=NT=2, 1≤NROWS, NCOLS≤3
MP	MP-matrix-vector.c [10]	-	170	7151	7905	7	921	25693	30	NP=3, 1≤NT=NROWS=NCOLS≤4
MP	MP-pie-collective.c [10]	+	79	23006	24723	12	1381	61623	44	NP=3, 1≤NITR≤5, NT=NITR/NP
MP	anl_hybrid.c [4]	-	43	27118	26932	10	1385	121099	4	NP=NT=2

Figure 7: Results of running CIVL `verify` command for C programs

disabled; an "a" indicates the full space of schedules is explored and an "r" indicates that round-robin thread scheduling is applied. Each "Example" program is described by name and with a citation for the source of the example, and the positive/negative verification result is reported as "+"/"-" in column "R". We measure the number of non-comment source lines, "LoC", for each example and provide the number of "States" and "Transitions" explored by the verifier, the "Time" required rounded to the nearest second, the "Mem"ory used in megabytes, and the number of validity calls, "ValidCalls", generated during verification and the number of calls that required invocation of an external prover, "Prove". Finally, parameters of programs that control the "Scale" of the program, e.g., its size, duration, number of threads or processes, are given. Since the verifier uses symbolic execution, constraints on parameters can be provided rather than simply fixing specific values.

All of the reported data was gathered by running release 1.4 of the CIVL toolset on an Apple iMac running OSX 10.9.2 (64 bit) with a 3.5 Ghz Intel core i7 processor. CIVL was configured to use Z3 4.3.2, CVC4 1.4, and CVC3 2.4.1, in sequence until a conclusive (not "unknown") answer is produced with a 10 second timeout for each prover call.

In most cases CIVL reported the properties to hold within the specified bounds. The exceptions include the SV-COMP Pthreads examples and some tutorial examples from LLNL which are intentionally defective; for these CIVL reported violating traces which we inspected and found valid.

CIVL found unintended errors in 8 programs and we confirmed, by manual inspection, that each reported error was an actual defect. Five types of errors were found: failure to declare variables private in OpenMP, a race condition in Pthreads, failure to call an MPI collective operation from all processes, reading uninitialized data, and failing to deallocate memory on particular schedules. For example, the LLNL OpenMP tutorial example `omp_bug5.c` has an intentional deadlock, which CIVL correctly reports. The tutorial proposes a fix, `omp_bug5fix.c`, but CIVL detects that this fix has a race condition, caused by a variable which should have been declared private. Changing the declaration allows CIVL to confirm the fix of the proposed fix.

6.2 Discussion and Future Directions

The data indicates the breadth of concurrency dialects that the CIVL toolset can support. Most of these programs were scaled down in order to permit them to be verified in a few minutes. CIVL is sufficiently scalable that configuration parameters could be set, to values of 3 or more, so that verification explored much of the complexity of a code base.

We did not attempt to sweep the scaling parameters to maximize them. In some cases, it is clear that very large parameterizations can be cost-effectively analyzed using CIVL. For example, a 100 interval instance of the MO hybrid "pie-calculation" example with a maximum of 10 threads represents significant complexity. This example divides the intervals across the 2 MPI processes and then uses an `omp for` to solve those intervals across up to 10 threads. Each of those `omp for`'s has more than 10^{50} possible schedules, but the OpenMP simplifier allows verification in just 7 seconds. In contrast, with just 6 intervals and 3 threads, verification without the simplifier requires 1412 seconds—a slowdown of 200 times relative to the simplified verification. While ef-

fective, the current OpenMP simplifier has significant room for improvement and we plan to explore extensions of it in future work.

Across the experiments reported in this study SARL is very effective in solving constraint queries. Of the nearly 24 million validity calls in our study, summing the ValidCalls column in Fig. 7, about 7000 required the invocation of an external prover. Thus, 99.97% of the calls were solved through simplification or caching within SARL rendering the prover time negligible in verification.

CIVL is novel in its support for multiple concurrency dialects. Single dialect tools can focus their optimizations on the semantics of that dialect and this may lead to better performance than CIVL. To assess this we compared the results of CIVL, TASS [50], and Lazy CSeq [19].

TASS is a highly-optimized verifier but it accepts a much smaller subset of C programs. We selected two examples, `diffusion1d.c` and `sum_array.c`, and simplified them by hand to work with TASS. The results were verified by TASS 1.1 in 10 and 2 seconds, respectively. TASS's focus on a single dialect allows these examples to run 4.5 and 6 times faster than CIVL. We believe, however, that some of the insights in TASS can be applied to further improve CIVL. For example, TASS allows a state to be mutable for efficient updates until it is ready to be stored on the stack, at which point it is "committed" and becomes immutable. This is a very effective and general optimization that could be used in CIVL (or other model checkers).

Lazy CSeq is a C/Pthreads verifier that won first place in the concurrency category of the 2014 SV-COMP competition. We ran version 0.6c (which won the competition) on the same Pthreads examples to which CIVL was applied. For 4 of the examples, CIVL and CSeq took the same amount of time, but for `queue_ok_longest_true-unreach-call.c`, CIVL and CSeq took 16 and 154 seconds, respectively. For `bug4.c`, CSeq returns `UNKNOWN`, while CIVL reports a *deadlock* with certainty `PROVABLE`, indicating the violation is guaranteed to be feasible.

These limited comparisons suggest that the generality of CIVL can be achieved with performance that is competitive with state-of-the-art dialect-specific verifiers. We plan to conduct a broader comparison across benchmarks and tools available from other researchers as future work.

7. REFERENCES

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

[2] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: Exploiting program structure for model checking concurrent software. In *Procedings of CONCUR*, pages 1–15, 2004.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall Europe, Herfordshire, UK, second edition, 1996.

[4] P. Balaji, J. Dinan, T. Hoefler, and R. Thakur. Advanced MPI programming. Tutorial at SC13: International Conference on High Performance Computing, Networking, Storage, and Analysis, Denver, Colorado, November 2013. Accessed Feb. 6, 2015.

[5] 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 N. Sharygina and H. Veith, editors, *Proceedings of CAV*, pages 863–868, 2013.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of CAV*, pages 171–177, 2011.

[7] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of CAV*, pages 298–302, 2007.

[8] C OpenMP examples. http://people.sc.fsu.edu/~jburkardt/c_src/openmp/openmp.html. Accessed Feb. 8, 2015.

[9] 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.

[10] Center for Development of Advanced Computing. Programming on Multi-Core Processors Using MPI - Pthreads. http://cdac.in/index.aspx?id=ev_hpc_hypack_mpi_pthreads_overview. Accessed Apr. 17, 2015.

[11] The Chapel parallel programming language. http://chapel.cray.com/. Accessed Feb. 8, 2015.

[12] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, Cambridge, Massachusetts, 2008. (examples).

[13] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of ICSE*, pages 439–448, 2000.

[14] CUDA Samples. http://docs.nvidia.com/cuda/cuda-samples/. Accessed Apr. 15, 2015.

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

[16] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Proceedings of TACAS*, pages 337–340, 2008.

[17] M. B. Dwyer, J. Hatcliff, Robby, and V. P. Ranganath. Exploiting object escape and locking information in partial-order reductions for concurrent object-oriented programs. *Form. Methods Syst. Des.*, 25:199–240, September 2004.

[18] J.-C. Filliâtre and A. Paskevich. Why3: Where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of ESOP*, pages 125–128, 2013.

[19] B. Fischer, O. Inverso, and G. Parlato. CSeq: A sequentialization tool for C. In N. Piterman and S. A. Smolka, editors, *Proceedings of TACAS*, pages 616–618, 2013.

[20] M. Flatt and PLT. The Racket reference, version 5.3.1. http://docs.racket-lang.org/reference/. Accessed Feb. 6, 2015.

[21] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of PLDI*, pages 212–223, 1998.

[22] M. Garland, M. Kudlur, and Y. Zheng. Designing a unified programming model for heterogeneous machines. In *Proceedings of Supercomputing*, Nov. 2012.

[23] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[24] 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*, 54(12):82–91, Dec. 2011.

[25] C. Hathhorn, M. Becchi, W. L. Harrison, and A. M. Procter. Formal semantics of heterogeneous CUDA-C: A modular approach with applications. In *Proceedings of the 7th Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012*, pages 115–124, 2012.

[26] G. J. Holzmann. *The* Spin *Model Checker*. Addison-Wesley, Boston, 2004.

[27] 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)*. IEEE, 3 Park Avenue, New York, NY 10016-5997, USA, Dec. 2008.

[28] International Organization for Standardization and International Electrotechnical Commission. ISO/IEC 989:2011 N1570: Programming Languages – C. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf, Apr. 2011.

[29] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In H. Garavel and J. Hatcliff, editors, *Proceedings of TACAS*, pages 553–568, 2003.

[30] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[31] Lawrence Livermore National Laboratory Message-Passing Interface (MPI) exercise. https://computing.llnl.gov/tutorials/mpi/exercise.html. Accessed Feb. 8, 2015.

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

[33] Lawrence Livermore National Laboratory Pthreads tutorial. https://computing.llnl.gov/tutorials/pthreads/exercise.html. Accessed Feb. 8, 2015.

[34] K. R. M. Leino. This is Boogie 2. http://research.microsoft.com/apps/pubs/default.aspx?id=147643, June 2008.

[35] 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 PPoPP*, 2012.

[36] P. Li, G. Li, and G. Gopalakrishnan. Practical symbolic race checking of GPU programs. In *Proceedings of Supercomputing*, pages 179–190, Nov. 2014.

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

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

[39] ParaSail Programming Language. http://www.parasail-lang.org. Accessed Feb. 7, 2014.

[40] C. S. Pasareanu and N. Rungta. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of ASE*, pages 179–180, 2010.

[41] Parallel programming course. http://users.abo.fi/mats/PP2014/examples/OpenMP/omp_critical.c. Accessed Feb. 8, 2015.

[42] W. Pugh and D. Wonnacott. Going beyond integer programming with the omega test to eliminate false data dependences. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):204–211, Feb. 1995.

[43] Purdue University, Information Technology: Research Computing. Carter – User Guide. https://www.rcac.purdue.edu/compute/carter/guide/#compile_gpu, 2008. Accessed Feb. 6, 2015.

[44] M. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[45] Z. Rakamarić and M. Emmi. SMACK: Decoupling source language details from verifier implementations. In A. Biere and R. Bloem, editors, *Proceedings of CAV*, pages 106–113, 2014.

[46] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.

[47] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley, 2010.

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

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

[50] S. F. Siegel and T. K. Zirkel. TASS: The Toolkit for Accurate Scientific Software. *Mathematics in Computer Science*, 5(4):395–426, 2011.

[51] R. M. Stallman and 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. Accessed Feb. 6, 2015.

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

[53] VirginiaTech: Advanced Research Computing. CUDA. http://www.arc.vt.edu/resources/software/cuda. Accessed Feb. 6, 2015.

[54] Zing language specification, Microsoft Corporation. http://research.microsoft.com/en-us/projects/zing/zinglanguagespecification.pdf, 2005.

[55] T. K. Zirkel, S. F. Siegel, and T. McClory. Automated verification of Chapel programs using model checking and symbolic execution. In G. Brat, N. Rungta, and A. Venet, editors, *Proceedings of NFM*, pages 198–212, 2013.