CIVL: The Concurrency Intermediate Verification Language

Stephen F. Siegel, Manchun Zheng, Ziqing Luo
Timothy K. Zirkel, Andre V. Manrianiello, 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.

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. 2.2). 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

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SC'15, November 15-20, 2015, Austin, TX, USA
© 2015 ACM, ISBN 978-1-4503-3723-6/15/11...
DOI: http://dx.doi.org/10.1145/2807591.2807635
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 ≥ 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 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
$\textbf{Scope}$ type for reference to a dyoscope; includes constants $\text{here}$ (the scope in which the expression occurs) and $\text{root}$

$\text{self}$ type: representing reference to an executing process; includes constant $\text{self}$

$\text{malloc}$ (scope, size): allocates object in heap of specified dyoscope; freed with $\text{free}$

$\text{for}$ (int i, j, ..., n) stmt: iterates over the tuples ⟨i,j,...⟩ in a domain d

$\text{choose}$ (n): expression returning an integer in [0, n - 1], chosen nondeterministically

$\text{choose}$ (a, b): creates and returns reference to new process executing function f

$\text{when}$ (guard) stmt: guarded command; enabled only when boolean expression guard evaluates to true

$\text{atomic}$ stmt: executes stmt without interleaving of other processes

Figure 2: Some commonly-used CIVL-C primitives

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 $\text{spawn}$ command returns an object of $\text{proc}$ type, which can later be used as the argument to $\text{wait}$. As mentioned in Sec. 2.2, each scope has its own heap. The function $\text{malloc}$ takes an extra argument of $\text{scope}$ type that specifies the heap in which memory should be allocated.

The $\text{domain}$ type and related functions make it easy to represent and manipulate “iteration spaces”. This is a common theme. An OpenMP $\text{for}$ 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 $\text{parfor}$ 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 $\text{wait}$, which is enabled only when the process specified by its argument terminates. Also, a guard can be attached to any statement using $\text{when}$. This can be used to program low-level concurrency constructs such as semaphores. Mechanisms for nondeterministic choice are provided by $\text{when}$ and $\text{choose}$.

Sevral primitives deal with specification. There are statements to $\text{assume}$ and $\text{assert}$ predicates, and first-order quantifiers $\text{forall}$ and $\text{exists}$. The $\text{input}$ and $\text{output}$ 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 $\text{bundle}$ 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 $\text{sequence}$ type supporting insert, delete, and append operations; a $\text{barrier}$ object with functions for creating, joining, invoking, and destroying barriers; and a $\text{communicator}$ type $\text{comm}$ 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 $\Sigma$ of static scopes. This has the structure of a rooted tree. The elements of $\Sigma$ correspond to the lexical scopes in the program: $\sigma$ is a child of $\sigma’$ if $\sigma$ is contained immediately (with no intervening scope) in $\sigma’$. 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 $\text{vars}(\sigma)$: the variables declared in $\sigma$. For each $\sigma$, $\text{vars}(\sigma)$ includes a $\text{heap variable}$ which is special in that it can be modified only through the system functions $\text{malloc}$ and $\text{free}$ (Sec. 2.1).

A model specifies a set of $\text{function symbols}$, which in-
includes 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 \( \Delta \) of dynamic scopes (or dyscopes), which also has the structure of a rooted tree, together with a tree homomorphism static: \( \Delta \to \Sigma \). If static(\( \delta \)) = \( \sigma \), we say \( \delta \) is an instance of \( \sigma \). For each \( \delta \in \Delta \), the state specifies a value for each variable in vars(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 \( \delta \) in state \( s \) if there exists a path from a dyscope occurring in \( p \)'s call stack to \( \delta \), following the parent edges in \( \Delta \). In the state of Fig. 3(right), for example, \( p3 \) reaches five dyscopes, with IDs \( 0, \ldots, 4 \). A dyscope \( \delta \) is reachable in \( s \) if there is some process which reaches \( \delta \) 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 \( \sigma \) to a new scope \( \sigma' \), a sequence of new dyscopes is added corresponding to the chain from the lowest common ancestor of \( \sigma \) and \( \sigma' \) to \( \sigma' \). 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 $\textbf{input}$ and $\textbf{output}$ 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 \texttt{civl run} executes the program by resolving all nondeterministic choices randomly. A random seed can be specified for reproducibility. Finally, \texttt{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 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 amale 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 amale 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 amale 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 \( \{ o_3, ..., o_9 \} \); \( p_1 \) reaches every object except \( o_2 \) (which is unreachable, and represents a memory leak); \( p_2 \) reaches \( \{ o_1, o_7 \} \). 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 \( o_7 \). 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.

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 \verb|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—though 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. 1, 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.

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 _mpi_init or _mpi_end, 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 libraries 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\), \$_thread_gpool\), and \$_omp_gteam\) 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. 6a), 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. 6c). OpenMP thread teams and shared variables are treated similarly; see lines 2-5, 7-8, and 10-12 in Fig. 6d). In each case, a small Enabler class is provided.

As illustrated in Fig. 6b), 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. 6b). Function nesting, as on lines 6-12 of Fig. 6b), 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 \$omp critical\) and \$omp barrier\), and mechanisms to control data sharing, such as \$omp private\) and \$omp 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. 6d) 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 \$omp 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 team threads, \$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-
Figure 6: Transformation examples (lightly edited for clarity)
tive array-dependence analysis by exploiting OpenMP data sharing dependencies and semantics to formulate constraints whose satisfiability assures the absence of loop-carried dependencies 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:

\begin{verbatim}
  for (int i=0; i<N; i++) a[i] = 0.0;
\end{verbatim}

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 \( k^w \).

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), CSQ [19] and TWinE 3.0 [5] (C/Pthreads), GLLIEE [33] 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 [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 [51] 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 [32]. 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/civil/sec15]. 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 (ROWS, NCOLS), number of time steps in simulations (NSTEPS), number 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. 4. And of course all of the standard and dialect-specific properties (Sec. 4) 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
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 a command for C programs.

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>R</th>
<th>LoC</th>
<th>States</th>
<th>Transitions</th>
<th>Time</th>
<th>Mem</th>
<th>ValidCalls</th>
<th>Prove</th>
<th>Scale</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>dotProduct.c</td>
<td>64</td>
<td>24</td>
<td>124</td>
<td>123</td>
<td>2</td>
<td>915</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>dotProduct.c</td>
<td>64</td>
<td>2366</td>
<td>2366</td>
<td>6</td>
<td>915</td>
<td>28248</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>matProduct.c</td>
<td>32</td>
<td>35</td>
<td>2656</td>
<td>266</td>
<td>6</td>
<td>915</td>
<td>471</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>heated_plate.c</td>
<td>44</td>
<td>160</td>
<td>228034</td>
<td>229308</td>
<td>33</td>
<td>1126</td>
<td>472742</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>fig3_10-mx-omp.c</td>
<td>12</td>
<td>62</td>
<td>821</td>
<td>820</td>
<td>2</td>
<td>915</td>
<td>2047</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>quad_openmp.c</td>
<td>4</td>
<td>91</td>
<td>8138</td>
<td>8138</td>
<td>11</td>
<td>915</td>
<td>2035</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>plt.c</td>
<td>44</td>
<td>70</td>
<td>44004</td>
<td>44088</td>
<td>10</td>
<td>915</td>
<td>54705</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>heated_plate.c</td>
<td>44</td>
<td>160</td>
<td>1684918</td>
<td>1700505</td>
<td>103</td>
<td>607</td>
<td>1135138</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>omp_bugs.c</td>
<td>32</td>
<td>54</td>
<td>2647</td>
<td>2652</td>
<td>4</td>
<td>916</td>
<td>3035</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>diffusion2d.c</td>
<td>19</td>
<td>274</td>
<td>489379</td>
<td>485418</td>
<td>247</td>
<td>915</td>
<td>257430</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>mpi_prime.c</td>
<td>41</td>
<td>105</td>
<td>28281</td>
<td>28276</td>
<td>14</td>
<td>915</td>
<td>78342</td>
<td>382</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>mpi_pi_send.c</td>
<td>41</td>
<td>120</td>
<td>112922</td>
<td>112357</td>
<td>101</td>
<td>732</td>
<td>365872</td>
<td>2421</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>sum_array.c</td>
<td>44</td>
<td>72</td>
<td>81852</td>
<td>81366</td>
<td>13</td>
<td>1393</td>
<td>430555</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>waveld.c</td>
<td>44</td>
<td>194</td>
<td>98091</td>
<td>97216</td>
<td>54</td>
<td>987</td>
<td>420943</td>
<td>240</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>wave1d Barrier</td>
<td>49</td>
<td>192</td>
<td>496</td>
<td>495</td>
<td>4</td>
<td>915</td>
<td>2118</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>gausselim.c</td>
<td>49</td>
<td>293</td>
<td>408185</td>
<td>408073</td>
<td>115</td>
<td>628</td>
<td>1796914</td>
<td>1911</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>matmat.c</td>
<td>49</td>
<td>104</td>
<td>85092</td>
<td>85294</td>
<td>18</td>
<td>1315</td>
<td>646793</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>cuda-omp.c</td>
<td>37</td>
<td>99</td>
<td>9401</td>
<td>10331</td>
<td>8</td>
<td>1409</td>
<td>442221</td>
<td>92</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>dot.c</td>
<td>47</td>
<td>99</td>
<td>13713</td>
<td>13921</td>
<td>6</td>
<td>915</td>
<td>110745</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>mm.c</td>
<td>13</td>
<td>146</td>
<td>877</td>
<td>878</td>
<td>3</td>
<td>915</td>
<td>3632</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>sum.c</td>
<td>37</td>
<td>72</td>
<td>12979</td>
<td>13144</td>
<td>3</td>
<td>915</td>
<td>15679</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>vectorAdd.c</td>
<td>37</td>
<td>148</td>
<td>4796</td>
<td>5179</td>
<td>5</td>
<td>650</td>
<td>695055</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>bug4.c</td>
<td>33</td>
<td>85</td>
<td>12162</td>
<td>12597</td>
<td>4</td>
<td>915</td>
<td>37915</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>queue_C longest...</td>
<td>92</td>
<td>126</td>
<td>68364</td>
<td>71574</td>
<td>16</td>
<td>915</td>
<td>121365</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>read_write_lock.</td>
<td>32</td>
<td>38</td>
<td>1756</td>
<td>1756</td>
<td>2</td>
<td>915</td>
<td>1702</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>sync01_true.c</td>
<td>52</td>
<td>42</td>
<td>320</td>
<td>329</td>
<td>1</td>
<td>915</td>
<td>602</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>i50_sincere_true</td>
<td>52</td>
<td>56</td>
<td>448</td>
<td>453</td>
<td>1</td>
<td>915</td>
<td>116</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>pio-calculation.c</td>
<td>5</td>
<td>65</td>
<td>622</td>
<td>619</td>
<td>6</td>
<td>915</td>
<td>1350</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>pio-calculation.c</td>
<td>5</td>
<td>66</td>
<td>65970</td>
<td>65970</td>
<td>7</td>
<td>915</td>
<td>5938</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>mpi_threads.c</td>
<td>33</td>
<td>87</td>
<td>32908</td>
<td>35778</td>
<td>10</td>
<td>915</td>
<td>923955</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>MP infinity-norm.c</td>
<td>33</td>
<td>146</td>
<td>2861</td>
<td>2896</td>
<td>6</td>
<td>915</td>
<td>6238</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>MP matrix-vector.c</td>
<td>33</td>
<td>170</td>
<td>7131</td>
<td>7905</td>
<td>7</td>
<td>915</td>
<td>25603</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>MP-pie-collective.c</td>
<td>10</td>
<td>79</td>
<td>23006</td>
<td>24723</td>
<td>12</td>
<td>1381</td>
<td>61623</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>anl_hyper.c</td>
<td>4</td>
<td>43</td>
<td>27118</td>
<td>26932</td>
<td>10</td>
<td>1385</td>
<td>121009</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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

6.2 Direction 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” parameterization with a maximum of 10 threads represents significant complexity. The example divides the intervals across the 2 MPI processes and then uses an omp for loop to solve those intervals across up to 10 threads. Each of those omp for’s has more than 10^9 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-
ffective, 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. 4 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, diffusionid.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


