Verification of MPI Programs using CIVL

Ziqing Luo, Manchun Zheng, and Stephen F. Siegel

Verified Software Laboratory
Department of Computer and Information Sciences, University of Delaware, USA
{ziqing|zmanchun|siegel}@udel.edu

Abstract. CIVL is a framework for verifying concurrent programs. The framework is built around a language, CIVL-C, that extends sequential C with general-purpose primitives that can be used to model a variety of concurrency dialects, including OpenMP, Pthreads, CUDA, and MPI. The framework automatically transforms programs using those dialects into CIVL-C so that static analysis and verification tools for CIVL-C can be applied. This paper describes how C/MPI programs are transformed and the hierarchy of support libraries used in the process. The result is a verifier that can check, within finite bounds, a number of difficult properties of MPI programs, including functional correctness, deadlock-freedom, and adherence to rules specified in the MPI Standard.

1 Introduction

The problem of verifying correctness of concurrent programs has bedeviled researchers since the advent of concurrency. Early efforts, e.g., [15], focused on deductive approaches (proofs) carried out manually. In recent years more automated approaches have been introduced, based not only on deduction but also static analysis, model checking, and symbolic execution. Today a number of tools exist for verifying various aspects of different categories of concurrent programs. While most are research prototypes, many have nonetheless shown promise by detecting numerous defects in and verifying—with certain caveats—a wide variety of concurrent programs (cf. [14]).

One of the main barriers to further progress is the sheer number of different ways of expressing parallel programs. MPI (the "Message Passing Interface" [13]), OpenMP, CUDA, and Pthreads are just a few of the "concurrency dialects" used to write modern parallel programs. New dialects are introduced regularly, and the old ones constantly evolve. Furthermore, modern parallel architectures have necessitated the use of multiple dialects within a single program; these *hybrid* programs are especially difficult to reason about. Since it takes enormous effort to develop a practical verification tool, almost all of these efforts have targeted a single dialect; very few have targeted hybrid programs. Moreover, these tools are rarely updated as the API Standards evolve. At the same time, many of the underlying verification techniques used in these tools are very similar, differing mainly in the dialect they support. This results in redundant work, inadequate tool comparisons, and slow progress.

The CIVL (Concurrency Intermediate Verification Language) framework was developed to address these challenges [20]. The framework is centered around a common concurrent programming language named CIVL-C. The front-ends translate C programs that use MPI, OpenMP, CUDA, and Pthreads (alone or in combination) to CIVL-C. The back-end used model checking and symbolic execution to verify CIVL-C programs. Ideally, a new dialect can be supported by just adding a new front-end; new verification techniques can be explored across a variety of dialects by adding a new back-end. In reality, some additional work is usually needed in each case to achieve reasonable performance. However, the language and framework have been designed to facilitate such extensions while keeping the work to a minimum.

A high-level description of the CIVL framework appeared in [20], but that paper did not go into detail on the translation from any specific concurrency dialect to CIVL-C. The purpose of this paper is to describe the translation process for one dialect — MPI. (Papers on the other dialects are in preparation.)

In Section 2, we summarize the aspects of the CIVL-C language—including its "core library"—relevant to the translation of C/MPI programs. The high-level structure of the translated code is explained in Section 3. Section 4 describes the general-purpose CIVL-C libraries used to model concurrency dialects, and Section 5 explains how those are used to model an abstract MPI implementation. Related work is discussed in Section 6. In Section 7, we evaluate the resulting tool in several ways: first, by enumerating desirable features of an MPI verification tool (Figure 4) and exploring how CIVL stacks up against other leading tools in its ability to provide these features; we also explore applications of CIVL to several realistic MPI applications; and finally we evaluate the usability and scalability of CIVL.

2 Summary of CIVL-C language

CIVL-C is an extension of the sequential part of C11 [7]. The names of the new keywords, types and functions all begin with `$`. The `$spawn` keyword may precede a function call, and indicates that a new *process* (or thread of control) should be created to execute the function call. The `$spawn` command returns immediately with a reference to the new process, an object of type `$proc`. The `$wait` function consumes an argument of type `$proc` and blocks until that process terminates.

CIVL-C allows function definitions to occur in inner (block) scopes, not just the file scope. Those functions can be spawned like any other function. This enables patterns such as the following:

```
1 void proc_f(int rank) {
2   void thread_f(int tid) {...}
3   $proc threads[nthreads];
4   $for (int i=0; i<nthreads; i++) threads[i] = $spawn thread_f(i);
5   $for (int i=0; i<nthreads; i++) $wait(threads[i]);
6 }
7 $proc procs[nprocs];
8 $for (int i=0; i<nprocs; i++) procs[i] = $spawn proc_f(i);
9 $for (int i=0; i<nprocs; i++) $wait(procs[i]);
```

The code above generates `nprocs` processes, each of which spawns `nthreads` "threads"—the basic structure of a hybrid MPI-threads program. Note the memory hierarchy implicit in the design: variables declared in the body of `thread_f` are "thread-local": they are accessible only by the single thread. Variables declared within the `proc_f` scope are "process-local": they can be accessed by the code of that process and all the threads spawned by the process. Variables declared in the outermost scope can be accessed by all threads and processes.

The pattern on lines 7–9 is so common that CIVL-C provides a convenient short-hand:

```
$parfor (int i: 0 .. nprocs-1) proc_f(i);
```

Scopes are first-class objects in CIVL-C. A value of type `$scope` represents a *dynamic scope*, an object created when control enters the '`{`' that opens the scope and disappears when control reaches the matching '`}`'. The keyword `$here` evaluates to the current scope, hence the statement

```
$scope proc_scope = $here;
```

if inserted between lines 1 and 2 above would give a name to the process scope. In CIVL-C, every scope has its own heap, and the `$malloc` function takes an extra argument specifying the scope in which memory should be allocated. Hence in this example it is possible for each process to use its own separate heap—a faithful model of a real MPI program.

CIVL-C has `$assert` and `$assume` statements, which have their usual meanings. There are universally and existentially quantified expressions (`$forall` and `$exists`), as well as `$lambda` expressions. The latter can be used to initialize an array. For example,

```
double a[n] = (double[n])$lambda (int i) i*2.0;
```

allocates an array of doubles of length n in which the i-th element is set to $2i$, for $0 \leq i < n$.

The type qualifiers `$input` and `$output` can be applied to variables in the global scope. An `$input` variable is read-only and is initialized to an arbitrary value of its type if no initializer is given. (The verifier initializes such a variable with a fresh symbolic constant.) An `$output` variable is write-only. Two programs with the same input/output signature can be *compared* by the CIVL verifier. This checks that on any execution, if given the same input, the two programs will produce the same output.

CIVL-C provides ways to extend the core language. Of course, ordinary functions are one extension mechanism. In CIVL-C, one can also add the function specifier `$atomic_f` to a function declaration. This means that a call to that function will happen in a single atomic step. In CIVL-C (as in C) the actual arguments are evaluated before the call, so their evaluation is not included in the atomic step; the step encompasses the call proper, the execution of the body, and the return. Such a call behaves very much like a new kind of statement.

CIVL-C also allows *system functions*. These are declared with the specifier `$system`, but definitions are not provided as CIVL-C code. Instead, the user provides a Java class which manipulates the state of the program directly. (The

idea is similar to SPIN's `c_code` facility.) System functions are always atomic. A system function may also have a *guard* which is specified in an annotation of the form `executes_when` *expr*`;` preceding the function declaration. A call to the function is enabled only in a state in which *expr* evaluates to *true*. By default, the guard is `$true`, i.e., a call is always enabled.

The problem with system functions is that verification tools have no idea what they can do. Without further information, a verifier must assume a system function could read or modify any part of the state. For model checkers, this can result in extreme state explosion. This is because model checkers use *partial order reduction* to reduce the state space explored without sacrificing soundness. POR requires knowledge of when two transitions are *independent*. (Two transitions are independent if, for any state in which they are both enabled, the state resulting from executing both is independent of the order in which they are executed.) Given a state s, for a model checker to restrict the search to a subset T of transitions enabled at s, it is necessary that on any execution departing from s, no transition dependent on a transition in T can occur without a transition in T occurring first [4].

A CIVL-C library developer can specify independence information for an atomic function with an annotation of the form `depends_on` *expr*`;`. One possible value of *expr* is `\nothing`, meaning a call to this function is independent of all transitions. Another is an expression of the form `\access`$(e_1,e_2,\ldots)$, where the e_i are expressions of pointer type. This annotation is understood as follows: given a state s, there is a directed graph G in which the nodes are the objects that exist in s and there is an edge from one node to another if the first object contains a pointer into the second. We say a process p can *access* an object o in s if there is a path in G from an object on p's call stack to o. The annotation specifies that the function call is independent of any transition executed by a process p that can *not* access any of the objects pointed to by the e_i. It is a fact of the CIVL-C language that if p cannot access o in s, then on any execution starting from s, p will never be able to access o unless some process that can access o executes first. Therefore, if there is some set of processes which cannot access the e_i objects, the model checker may be able to ignore the transitions from those processes. (There are other conditions that must be checked, but they do not require annotations.) Recall that the evaluation of actual arguments takes place before the function call; the dependence claim covers only the call, execution, and return—all of which happens as a single atomic step.

A dependence clause may be provided for any atomic function, not just system functions. This is useful when the independence holds for some subtle reason that the model checker cannot determine on its own. Of course, it is possible for a dependence clause to be wrong, i.e., for a function to depend on more than what is specified by the clause. In that case, the CIVL verifier could conclude that a property holds when it does not. However, this mechanism is intended only for "expert developers"—e.g., those that are developing a new front-end—not the end users. Moreover, it is also possible that independence relations that are hard-coded into model checkers are buggy. Placing this information in the

source code makes transparent the assumptions about independence. Finally, in some cases unsoundness may be desirable, for example, in bug-finding tools.

3 Transformation

The CIVL front-end preprocesses, parses, and merges source files to create a single AST representing the whole program. For each concurrency dialect, an AST-to-AST transformation then replaces dialect-specific code with equivalent CIVL-C code. In this section we describe the MPI transformation. But first we give a brief overview of MPI.

In threading dialects, a program typically starts with a single thread of control and then explicitly creates and destroys threads. MPI is different. A complete MPI program consists of the code that will be run by a single process. Conceptually, the program is executed by duplicating that code n times (where $n \geq 1$ is specified when the program is launched), with each copy running in its own process. The global variables in the original program essentially become process-local variables. All inter-process communication and synchronization is carried out by explicit calls to functions in the MPI library.

A *communicator* c is an MPI abstraction representing a "communication universe". Conceptually, c comprises an ordered set of m processes ($m \geq 1$). The processes are numbered from 0 to $m - 1$; that number is the *rank* of the process in c. A process may belong to multiple communicators, and have a different rank in each one. However, MPI provides one default communicator, `MPI_COMM_WORLD`, which consists of all processes at system startup. The rank in `MPI_COMM_WORLD` can be thought of as a PID. By branching on this PID, process behavior can diverge in arbitrary ways. MPI provides many functions to produce new communicators from old ones.

A process refers to a communicator using an opaque handle, which is an object of type `MPI_Comm` (e.g., `MPI_COMM_WORLD`). All communication operations take an `MPI_Comm` argument. Messages sent on a communicator can only be received using that communicator. Conceptually, a communicator of size m encompasses m^2 FIFO channels—one for each ordered pair (i, j)—for buffering messages sent from rank i to rank j. It is also possible to break the strict FIFO ordering by associating integer *tags* to messages and specifying a tag in the receive call.

In addition to the usual *point-to-point* operations for sending and receiving messages, MPI provides a number of *collective* operations that involve all processes in a communicator: barriers, broadcasts, etc. The collective and point-to-point buffers are completely disjoint.

Figure 1 shows the structure of the CIVL-C translation of an MPI program. The transformation introduces input variables for the number of MPI processes, as well as lower and upper bounds on that number. The CIVL verifier allows the user to specify concrete values for input variables on the command line, so one may verify the program, for example, for any number of processes between 2

```
1 $input int _mpi_nprocs, _mpi_nprocs_lo = 1, _mpi_nprocs_hi, _civl_argc;
2 $input char _civl_argv[_civl_argc][];
3 $scope _mpi_root = $here;
4 $assume(_mpi_nprocs_lo <= _mpi_nprocs && _mpi_nprocs <= _mpi_nprocs_hi);
5 $mpi_gcomm _mpi_gcomm_world, _mpi_gcomms[]; // global communicators
6 void _mpi_process(int _mpi_rank) {
7   $mpi_state _mpi_state = _MPI_UNINIT; // also: _MPI_INIT, _MPI_FINALIZED
8   MPI_Comm MPI_COMM_WORLD= $mpi_comm_create($here, _mpi_gcomm, _mpi_rank);
9   int MPI_Send(void * buffer, int count, MPI_Datatype datatype,
10        int dest, int tag, MPI_Comm comm) {
11    $assert(_mpi_state==_MPI_INIT, "can't call MPI_Send before MPI_Init");
12    $mpi_check_buffer(buffer, count, datatype);
13    return $mpi_send(buffer, count, datatype, dest, tag, comm);
14  }
15  [more definitions of MPI functions]
16  [insert original source code here, but rename main _civl_main]
17  // invoke original main function...
18  _civl_main(_civl_argc, (char * [_civl_argc])$lambda (int i) _civl_argv[i]);
19  $mpi_comm_destroy(MPI_COMM_WORLD);
20 }
21 int main() {// initialize global communicator and launch processes
22  _mpi_gcomm_world = $mpi_gcomm_create(_mpi_root, _mpi_nprocs);
23  $seq_init(&_mpi_gcomms, 1, &_mpi_gcomm_world);
24  $parfor (int i: 0 .. _mpi_nprocs - 1) _mpi_process(i);
25  $mpi_gcomm_destroy(_mpi_gcomm_world);
26 }
```

Fig. 1: Translation of C/MPI program to CIVL-C: high-level structure

and 10. (The default lower bound is 1). The command line arguments (not used in this example) are also represented as input variables.

An object of type `$mpi_gcomm` (Section 4) is a *global communicator object*. This represents the global state of an MPI communicator, including its sequence of member processes, and all of their buffered messages. The variable `_mpi_gcomm_world` stores a reference to the global communicator object for `MPI_COMM_WORLD`. The array `_mpi_gcomms` stores references to all communicators; initially it contains only `_mpi_gcomm_world`. These global communicator objects are allocated in the global scope (shared by all processes), and are essentially the only state data in that scope.

The entire original program is inserted into a function `_mpi_process` (lines 6–20). A process-local variable is introduced to record the "state" of the process, which starts as *uninitialized*, changes to *initialized* when `MPI_Init` is called, and then to *finalized* when `MPI_Finalize` is called. The handle `MPI_COMM_WORLD` is also allocated in the process scope; it wraps a reference to the global communicator object with the rank. CIVL's models of the MPI functions are all inserted in this scope as well, so they can access the PID and other process-local data. The original *main* function is renamed `_civl_main` and is invoked on the command line arguments. Finally, a new main function is introduced in the global scope. It initializes the global communicators (lines 22–23), spawns and waits for the processes, each of which runs the `_mpi_process` function with a unique rank between 0 and `_mpi_nprocs-1` (line 24), and deallocates the global communicator upon termination (line 25).

4 General utility libraries

The CIVL utility libraries (Figure 2) provide general-purpose procedures and abstract data types that can be re-used to model multiple concurrency dialects. We describe these five libraries here. All of the types and most of the functions are defined in CIVL-C code, the remaining are system functions. Most of the functions are also atomic and have associated guards and/or dependency clauses.

4.1 `pointer.cvh`: general pointer utilities

The CIVL-C memory model encodes more information than is required to implement C. First, the state of a CIVL-C program comprises a set of values, and the values are *typed*. Second, a pointer value is a reference to a logical point in the hierarchical structure of an object, such as "field 3 of element 17 of o", where o is some array of structs. This logical model permits all C operations that do not result in undefined behavior according to the C Standard,[1] including bit-wise operations, pointer arithmetic, and casts. It also enables some operations that are not possible in C but are extremely convenient for modeling.

This library provides functions operating on void-pointers that take advantage of the CIVL memory model. The `$copy` function, for example, consumes two pointers p and q, and copies the logical (sub-)object pointed to by q to the location specified by p. The types of the values at these two memory locations must be compatible or an error is reported. Note there is no need to specify a "size" argument, as there is for C's `memcpy`.

This library also provides the `$apply` function to apply a binary operation to two objects and store the result at a specified location. This is used in the model of MPI's reduction operations.

4.2 `seq.cvh`: mutable sequences

Another CIVL-C language extension is the ability to declare and use variables of incomplete array type, i.e., without specifying the length of the array. Such variables may be assigned array values of varying length, much like a "vector" in other languages. In CIVL-C these are called *sequences*, and this library provides functions to manipulate them. There are methods to append, remove, and insert elements, and to get the current length of a sequence. The usual array index syntax can be used to read or write an element of a sequence. Sequences are used extensively in the MPI model, for example to model message queues.

4.3 `bundle.cvh`: data bundles

CIVL's MPI model uses a sequence to represent a FIFO channel. This array, like every state component, must have a type. What is the type of a message? The

[1] For example, pointer arithmetic that goes beyond the bounds of an object results in undefined behavior and will be reported as an error by CIVL.

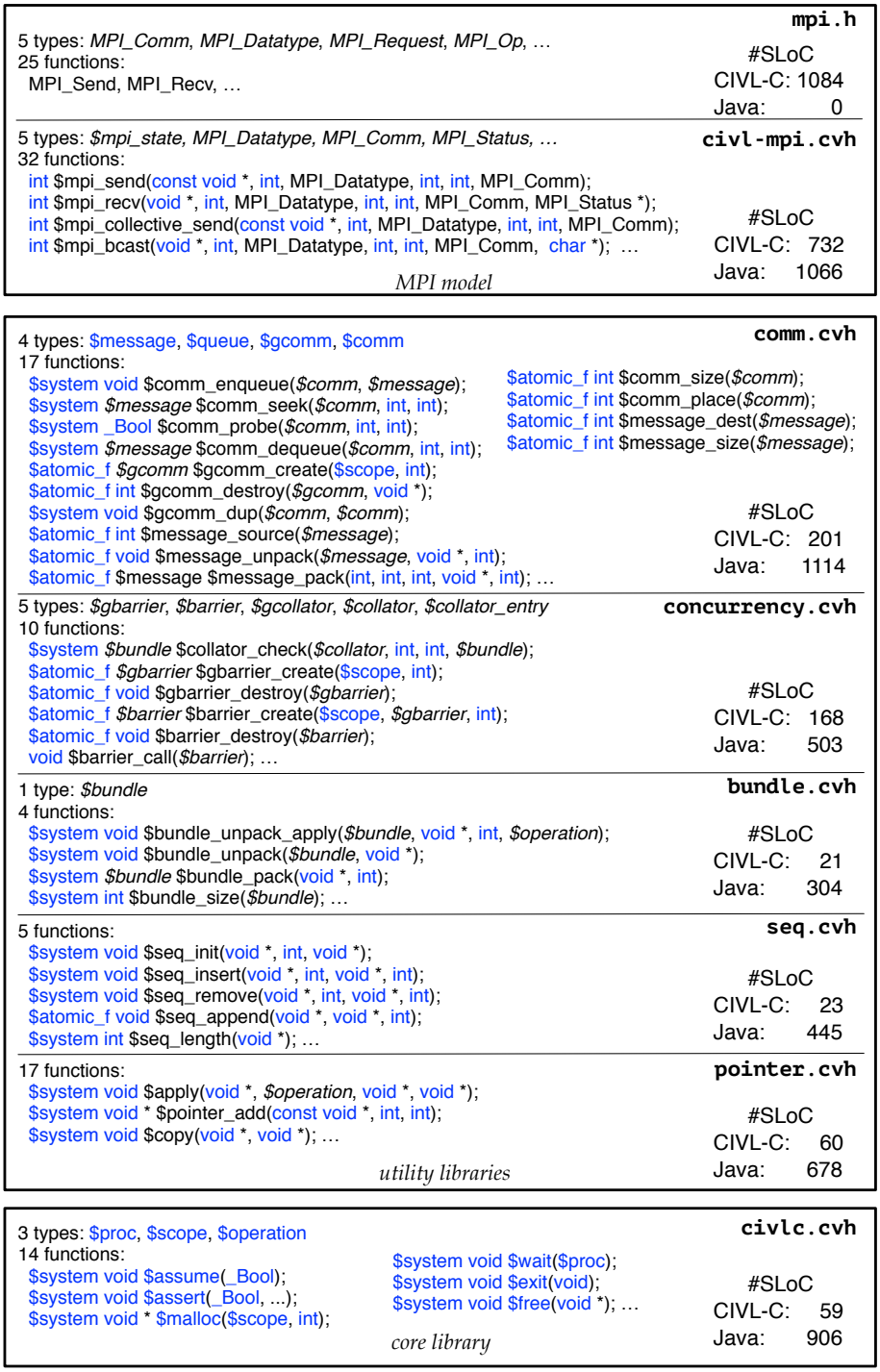

Fig. 2: CIVL-C libraries used in translating MPI programs, including numbers of non-comment source lines of code for their implementation

data component can vary unpredictably: one message may consist of a sequence of `int`s, the next a sequence of `double`s, the next a sequence of structures. It is not obvious how to capture all of these under a common type.

For this reason, CIVL-C provides a *bundle* type. A bundle "wraps up" a region of memory into a single value. The region must consist of some sequence of values of a single type. The function `$bundle_pack` consumes a void pointer and a size argument specifying the region, and returns an object of type `$bundle` that may be thought of as a snapshot of the state of the specified region. Bundles are immutable. A dual function unpacks a bundle into a specified memory region.

4.4 `concurrency.cvh`: concurrent data structures

This library provides general-purpose concurrent data structures which are used to model multiple dialects. One such structure is a reusable *barrier*—an object used to synchronize a set P of processes. A *global barrier object* is created by `$gbarrier_create`. This function consumes the scope in which the data structure should be allocated, and the size of P. It returns an opaque handle (implemented as a pointer to the global barrier object) of type `$gbarrier`. Each process $p \in P$ then "registers" once with the global barrier by invoking `$barrier_create` on the global handle and a unique integer PID in $[0, |P| - 1]$. This function returns an opaque handle of type `$barrier` which is used by p in its calls to `$barrier_call`—the actual barrier function.

This pattern—a handle to a shared "global" object containing all the state data, together with a process-local handle for each process—is used repeatedly in CIVL. This facilitates precise specification of independence. Many functions that access the shared structure commute for nontrivial reasons, e.g., two calls to enter a barrier from different processes, or two invocations of "send" on a communicator from different processes. Such functions can be annotated with "`depends_on \access(`h`);`", where h is the process-local handle parameter.

Another useful structure is the *collator*: an object that is used to collect information from each process participating in some collective action. A process creates an *entry* each time it passes through the collator, and that entry is enqueued in a FIFO queue for that process. As soon as there is at least one entry in each queue, one entry is dequeued from each queue and a checking function is applied to those dequeued entries. This abstraction has many uses in OpenMP as well as MPI. For example, it is used to check that all MPI processes belonging to a communicator make the same sequence of collective calls, that those calls agree on the "root" argument, the type and size of data, and so on.

4.5 `comm.cvh`: data structures for message-passing communication

This library provides a basic message-passing model. It defines a *message* type which is a structure comprising a bundle, and integers for a *source*, *destination*, and *tag*. A *communication object* comprises n^2 FIFO channels of messages for a fixed set of n "places" (which may, for example, correspond to MPI processes). Functions to dequeue, enqueue, and probe messages are provided. Following

```
typedef struct MPI_Comm {
  $comm p2p; // point-to-point communication
  $comm col; // collective communication
  $collator collator; // used to check collective call consistency
  $barrier barrier; // used to implement MPI_Barrier
  int gcommIndex; // index of corresponding global communicator
} MPI_Comm;
int $mpi_send(void *buf, int count, MPI_Datatype datatype,
     int dest, int tag, MPI_Comm comm) {
  if (dest >= 0) {
    int size = count*sizeofDatatype(datatype);
    int place = $comm_place(comm.p2p);
    $message out = $message_pack(place, dest, tag, buf, size);
    $comm_enqueue(comm.p2p, out);
  }
  return 0;
}
```

Fig. 3: CIVL-C definitions of ***MPI_Comm*** and **$mpi_send**

the usual pattern, there is a single global communication object containing all of the state data; a handle to this object has type ***$gcomm***; the local handle corresponding to a single place has type ***$comm***.

Communication objects are used to model MPI communicators. In fact each MPI communicator is a structure containing two such objects: one for point-to-point communication, the other for collective communication.

5 Implementation of the MPI library

The MPI library is implemented using all of the tools described above. A support library `civl-mpi` defines some of the basic types and auxiliary functions. The final implementation of `mpi.h` uses that support library, and is pure CIVL-C code (no system functions).

As explained in Section 4.5, MPI communicators are implemented using the ***$gcomm***/***$comm*** structures of the `comm` library. Figure 3 shows the definition of ***MPI_Comm***. This structure has fields for two local comm handles (one for point-to-point and one for collective messages), a collator for checking consistency of collective calls on the communicator, a barrier, and the index of the communicator in the global list of communicators.

Figure 3 also shows the auxiliary function used to send a point-to-point message. A negative destination value is used to represent `MPI_PROC_NULL`; sends to that destination are no-ops. Otherwise, the size of the send buffer is computed, the rank of the calling process in `comm` is obtained, and a message is created and enqueued in the point-to-point section of the communicator. Finally, **MPI_Send**, shown in Figure 1, simply checks that the process has been initialized and that the send buffer contains values of the appropriate type, and then calls this auxiliary function.

The collective functions are defined using auxiliary send/receive operations that access the collective section of the communicator. Simple, deterministic algorithms are used, unlike the case in a real MPI implementation.

6 Related Work

There has been much research on verification of MPI programs. Some of the earliest work (e.g., [12,18,19]) used the model checker Spin. While Spin's input language, Promela, provides some general concurrency primitives, using the language to model the complexities of an abstract MPI runtime is non-trivial. In that sense, the focus of the early work is similar to that of this paper. The main difference is that Promela is far removed from the C language, so it requires significant manual effort to abstract and transform a C/MPI program into a Promela model. This was ameliorated somewhat in MPI-Spin [18]—an extension to Spin which defines macros corresponding to many of the MPI functions. These functions include the blocking-mode standard point-to-point functions, all the *nonblocking* functions, and the collective functions—a larger subset of MPI than that covered by CIVL. MPI-Spin also provides a primitive symbolic execution facility which has been used to verify functional equivalence of MPI programs with corresponding sequential versions. Still, without pointers, procedures, and many other basic C constructs, producing a useful Promela model of a C/MPI program is difficult and error-prone.

Like Spin, TASS [22] is an explicit-state, stateful model checker, but its input language is a subset of C and MPI. Using symbolic execution, it can verify that properties hold for all possible inputs to a program, within some specified bounds on input sizes; it can also show that two programs are functionally equivalent. However it is MPI-specific and cannot be applied to other concurrency models or hybrid programs. It supports only a very limited subset of C.

ISP [25] and its distributed cousin DAMPI [26] are dynamic model checkers for MPI programs. They use a modified MPI runtime to explore schedules and communication matchings in a systematic way, and can verify properties such as deadlock-freedom. These tools do not perform symbolic execution, so require specific concrete inputs, similar to testing. They are stateless, so each interleaving is executed completely from beginning to end, which can reduce performance relative to a stateful model checker. On the other hand, these tools are quite robust and require no restrictions on the input language.

MOPPER [5] verifies deadlock-freedom in *single-path* MPI programs, including those that use (wildcard) `MPI_ANY_SOURCE` receives. It uses ISP to generate a single trace of the program and then reasons about all possible wildcard matchings that could occur in that trace. It does this very efficiently by encoding the problem as a SAT formula. In our experience, however, it is rare that for a program using `MPI_ANY_SOURCE` to satisfy the single-path restriction. e.g., in a manager-worker pattern, the manager will receive a task from any worker using a wildcard, and then send the next task back to that worker—violating single-path. On such programs, MOPPER's analysis is not necessarily sound.

There are also a number of dynamic checkers for MPI programs, such as MUST [6]. These are the most scalable of all the approaches discussed so far, but are limited to verifying a single run of a program, on concrete inputs.

All of the tools discussed above require a fixed or bounded number of MPI processes, and often bounds on other parameters (such as the size of input ar-

rays). There has been some research into verifying properties for programs of arbitrary scale. Dataflow analyses on "parallel control flowgraphs" that generalize the traditional notions for sequential programs are one approach [1,3,17,23]. While this is an active field of research, we are not aware of any publicly available tools based on it.

Another promising unbounded approach is the use of "session types" and automated theorem proving techniques to verify deadlock-freedom of MPI programs [10, 11, 16]. ParTypes [10] takes this approach. These tools currently require a good deal of manual effort: the user must provide a formal description of the communication pattern used in the MPI program, and insert annotations in the code linking program elements to that description. ParTypes also does not support wildcards (or use of multiple tags), or nonblocking operations. Hence it is restricted to a determinisitic subset of MPI.

The idea of representing a variety of concurrency dialects in a single generic language has also arisen in the compiler research community. The INSIEME compiler framework uses an intermediate representation called INSPIRE [8]. Like CIVL, INSPIRE is based on a general concurrency model, and includes primitives for spawning thread groups, nesting groups, and channel-based communication. Like CIVL, the Clang-based INSIEME front-end translates C programs using a variety of concurrency dialects to INSPIRE. Unlike CIVL, the goals are those of a traditional compiler—optimization and code-generation, and a runtime system—not verification. Moreover, INSPIRE is more "low-level" than CIVL-C, more comparable to the CIVL "model" IR [20]. CIVL-C is intended to be not only an IR, but a language that people will like to read and write directly. For example, a member of the MPI Forum might use it to prototype and explore new MPI features, it could be used to teach a class on model checking, and so on. Finally, while [8] includes discussion of mapping MPI programs to INSPIRE, at the time of writing INSIEME supports OpenMP and OpenCL but not MPI, so a detailed comparison is not possible. The framework also includes a formal language for specifying transformations of ASTs, which allows for very succinct encoding of transformations [9].

7 Evaluation

We evaluated CIVL (v1.7.2) in three ways: (1) by performing a "feature comparison" with four leading MPI verification tools—MPI-Spin (v1.0), ISP (v0.3.1), MOPPER (based on ISP v0.2.0), and ParTypes (v1.0.3); (2) by applying CIVL to 5 more realistic MPI applications; and (3) by performing two scaling experiments involving all of the tools. Our testbed is an iMac with a 3.50 GHz Intel Quad Core i7-4771 CPU and 32 GB memory. ParTypes was run on a 64-bit Windows 7 VM with 1 GB memory on the testbed (since it only works with Windows); MOPPER was run on a 64-bit Ubuntu 14.04.5 VM on the testbed; the other tools were run on the testbed directly. All artifacts, including source programs, tool output, and instructions for reproducing the experiments, can be downloaded from `http://vsl.cis.udel.edu/civl/vmcai17/`.

	Property and test programs	**C**	**S**	**I**	**M**	**P**
	An ideal verifier should verify the absence of ...					
DL	deadlocks: `absolute_dl`, `potential_dl`	+	+	+	+	+
AS	assertion violations: `assertion`	+	+	+	-	*
RO	messages that overflow the receive buffer: `rbuf_overflow`	+	+	-	-	+
TM	incompatible message and receive types: `type_mismatch_p2p`	+	+	-	-	-
CM	inconsistency in collective calls: `collective_mismatch`	+	+	+	+	+
FE	failure of an MPI program to be functionally equivalent to a sequential version of that program: `matmat`, `matmat_mw`	+	+	-	-	-
SE	divisions by zero, reads before definition, out of bound indexing, and illegal pointer dereferences: `seq`*`1..4*	+	-	-	-	-
	An ideal verifier should reason soundly about ...					
UB	arbitrary (unbounded) number of processes: `parallel_dot`	-	-	-	-	+
IN	all program inputs (of bounded size): `input_branch`	+	+	-	-	+
DD	control paths with dependencies on message data: `data_depend`	+	+	+	+	-
DB	dynamically changing blocking choices for `MPI_Send`: `dy_buf`	+	+	-	-	-
AR	all matchings at `MPI_ANY_SOURCE` receives: `any_src`	+	+	+	+	-
MP	control paths resulting from wildcard matchings: `not_single_path`	+	+	+	-	-
MT	multiple tags and `MPI_ANY_TAG`: `tags`	+	+	+	+	-
NB	non-blocking operations: `simple_nb`	-	+	+	+	-
MC	multiple communicators: `comm_dup`	+	-	+	-	-
HY	multithreaded MPI ("hybrid") programs: `mpithreads`	+	-	-	-	-
	An ideal verifier should ...					
US	be easy to use (automation): `parallel_dot`, `matmat_mw`	+	-	+	+	-
SC	scale well: `parallel_dot`, `matmat_mw`	*see §7.3*				

Fig. 4: Desirable features of a tool for verifying MPI programs and corresponding tests; results of running **C**IVL, MPI-**S**pin, **I**SP, **M**OPPER, **P**arTypes.

7.1 Feature tests

In Figure 4, we list some of the most important features an MPI verification tool should provide. These include the kinds of properties the tool should check (or dually, the kinds of defects it should detect); the *scope* of program behaviors and constructs about which it should reason soundly; and usability and scalability. Working from this list, we constructed a suite of simple programs to test each feature. Each of these feature tests has at least one correct version and one incorrect version exhibiting a specific defect. We applied the five tools to each test program and used the results to conclude whether a tool provides a feature. For MPI-Spin, we constructed Promela models by hand; for ParTypes, we wrote appropriate protocols and added annotations; FE required a small amount of CIVL annotation.

As can be seen from Figure 4, CIVL supports a wide range of features. There are two main reasons for this: (1) the use of symbolic execution, and (2) the high fidelity of the translation from C/MPI source to the CIVL-C model. Consider, for example, TM. The symbolic execution mechanism associates a dynamic type

to every value. When a bundle (Sec. 4.3) encoding message data is unpacked, the type of the data is compared to that specified in the receive; if an inconsistency is discovered, an error is reported. Symbolic execution also enables features FE, SE, IN, and DD. The ***$comm*** data structure (Sec. 4.5) encapsulates complete information about the state of buffered messages: their sources, destinations, tags, contents, and FIFO ordering; multiple communicators are represented using a list of ***$comm***s that can be created and destroyed dynamically, delivering feature MC. The symbolic execution approach does entail certain limitations: the number of processes must the sizes of input data structures such as arrays must be bounded. Moreover, CIVL does not yet support nonblocking operations.

In contrast, ParTypes can verify programs for any number of processes, but it does so for only a very limited subset of MPI. In particular, it does not deal with any-source (wildcard) receives, so the MPI programs it accepts are deterministic: given any input and number of processes, deadlock-freedom and assertion violations can be verified by examining a single synchronous execution. Verifying deadlock-freedom in programs with any-source receives is much more challenging, because multiple matchings and execution paths must be explored. Moreover, ParTypes has limited ability to reason about the content of messages. For example, our test `data_depend.c` (in pseudocode) is

```
int a[2];
if (rank == 0) {a[0]=1; a[1]=2; send(1,a);}
if (rank == 1) {recv(0,a); if (a[0]!=1 || a[1]!=2) recv(0,a);}.
```

This program is deadlock-free (since the condition in the `if` statement must be *false*) but we could not find any protocol which allowed ParTypes to verify it.

An MPI program deadlocks "absolutely" when every process is either terminated or blocked at a receive for which no matching send or message is present, and at least one process has not terminated. However, this is not the only way deadlock can occur: a program in which every process is either terminated, blocked at an unmatched receive, or blocked at an unmatched **`MPI_Send`**, *may* deadlock. This is because any invocation of **`MPI_Send`** may be forced to synchronize (i.e., block until a matching receive operation is posted) or may be executed by buffering the message. These "potentially" deadlocked states are a superset of the absolutely deadlocked states and are particularly hard to detect.

All of the tools correctly detect simple examples of absolute and potential deadlock. However they diverge on more complicated examples, such as `dy_buf_bad.c`. This program, which uses an any-source receive, contains a potential deadlock which can only be reached if one send is forced to synchronize and another is buffered. ISP and MOPPER cannot detect this deadlock. This is because they use a *fixed-capacity* channel model: the user may specify that every send must be synchronous, or that every message channel has a fixed positive number of messages it may hold. In contrast, MPI-Spin has a mode in which it makes a nondeterministic choice at each send—to either buffer or force synchronization. This is sound, but scales poorly. CIVL does better: in potential-deadlock-detection mode, it explores all possible states with infinite buffering, but checks for potential deadlocks at each state. The POR scheme is adjusted

to be sound for potential deadlock, which means more states are explored than for absolute deadlock, but not as many MPI-Spin explores.

ISP and MOPPER require concrete inputs. If a defect only manifests on certain inputs, they cannot detect the defect unless executed on one of those inputs. In this regard, their approach is closer to testing, though unlike testing, they explore a wide range of possible executions for the given input. CIVL and MPI-Spin use symbolic execution and thus can reason about a wide range of inputs.

One measure of *lack* of usability is the amount of annotations or modeling code that must be written. Figure 5 gives the number of lines of additional such code for two examples. MPI-Spin is the worst by this measure, as it requires the user to write a complete Promela model of the program. ParTypes requires a protocol description and annotations in the original MPI program for matching C constructs and protocol types. CIVL requires only a small amount of extra code to specify bounds on the number of processes, and (depending on the problem) to specify input data. ISP and MOPPER are the best in this regard and required only 3 lines of annotation for specifying input data.

Another aspect of usability is the ability to provide useful feedback when something cannot be verified or a defect is discovered. MPI-Spin is based on Spin and thus provides traces as counterexamples, but the user must figure out how they relate to the original program. ParTypes reports errors as VCC assertion violations, without explanation. ISP, CIVL, and MOPPER provide step by step traces through the original source code; CIVL additionally displays the path condition, and the (symbolic) values of all variables involved in an assertion violation, or the call stacks of each process in a deadlock.

7.2 Verifying realistic MPI programs using CIVL

In Figure 6, we present the result of using CIVL to verify the selected five real-world programs: 2d-diffusion, matrix multiplication using a manager-worker pattern, an MPI-Pthreads implementation of dot-product, Gaussian elimination on a row-distributed matrix, and a 1d-wave simulation with a subtle defect. For each program, CIVL checks its complete set of standard and MPI-specific properties as shown in Figure 4, including the validity of all assertions, which often involve complex numerical computations. In addition, for those marked with *c*, functional equivalence with a separate, trusted sequential version was also verified.

We use + to denote absence of property violations, and - for the opposite. We record the number of states and transitions explored by CIVL. The scale of each

Example (lines of code)	CIVL	ParTypes	MPI-Spin	ISP/MOPPER
`parallel_dot.c` (73)	3	63	98	3
`matmat_mw.c` (63)	13	N/A	350	0

Fig. 5: Annotation burden measured by number of lines of code

program is controlled by the number of processes NP and other input parameters, such as NSTEPS, which represents the number of time steps in a simulation.

CIVL was able to obtain the expected result (+ or -) on a range of configurations of these benchmarks. On the most realistic examples, the number of processes used was 5, and in the worst case time was 6 minutes: this occurred for the highly nondeterministic manager-worker matrix multiplication program, for 8×8 matrices and 5 processes.

7.3 Scalability

We analyzed how the tools scale by the number of processes (NP) while verifying deadlock-freedom for two examples, dot product and matrix multiplication, as shown in Figure 7. For the dot product program, which is deterministic, ISP and MOPPER are much faster than CIVL (but CIVL does not require concrete input vectors). ParTypes verifies MPI programs against arbitrary NP, so it has constant performance. MPI-Spin executes transitions very quickly, but suffers from an extreme state explosion; this is due to the inserted C code in its model of the MPI library. Spin has no knowledge about the inserted code and must assume that it can access anything, and is therefore dependent on all other transitions, defeating Spin's POR. That explains why MPI-Spin becomes slower than CIVL when $\texttt{NP} > 7$ and cannot complete within 30 minutes when $\texttt{NP} = 10$.

The matrix multiplication example uses wildcard receives and is not single-path, so ParTypes and MOPPER cannot be applied. Of the remaining three tools, CIVL is the slowest over most of the domain. However, for $\texttt{NP} \geq 6$, CIVL's time begins to decrease. This reflects the combinatorial nature of the problem: the initial NP tasks are sent to workers in a deterministic order, so as NP approaches the number of tasks (8), the amount of nondeterminism decreases. (There is still nondeterminism because the tasks can be returned in any order.) Also, for $\texttt{NP} = 6$, ISP cannot complete within 30 minutes. We believe the difference is due to the fact that CIVL is a stateful model checker—it saves the seen states and can backtrack as soon as a state has been seen before—while ISP is stateless. MPI-Spin also saves states but is hampered by the ineffective POR scheme. CIVL takes much longer to execute a transition, but uses a very precise POR algorithm that is aware of the semantics of each MPI function. It therefore explores significantly fewer states than MPI-Spin, and by $\texttt{NP} = 9$ is actually faster than MPI-Spin.

program	SLOC	result	states	transitions	time(s)	scale
diffusion2d.c [21]	276	+	699656	690734	201	NP=4, NSTEPS,NX,NY∈[1,5]
matmat_mw.c [21]^c	89	+	694447	693636	357	NP=5, N=L=M=8
mpithread_both.c [2]	96	+	109868	127027	28	NP=MAXTHRDS=2, VECLEN=5
gauss_elim.c^c [21]	295	+	287641	285584	93	NP=ROW=COL=3
wave1dBad.c [20]	190	-	505	504	3	NP∈[1,4], NSTEPS,NX∈[1,5]

Fig. 6: Results of verifying C/MPI programs using CIVL. Result: + (all properties hold), or - (violation found).

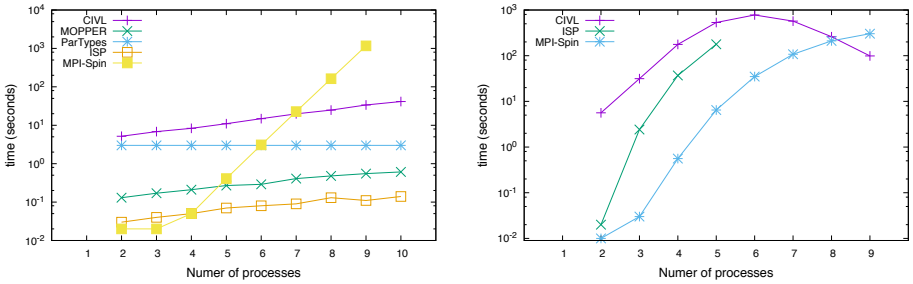

Fig. 7: Time to verify deadlock-freedom. Left: dot product, vector length 50. Right: manager-worker matrix multiplication, 8×8 matrices.

8 Conclusion

MPI and the C programming language are large and complex languages/APIs. Clearly any attempt to automatically translate C/MPI programs into the language of a generic verification tool will be a challenge. Nevertheless, we have found that CIVL manages this challenge reasonably well. By judicious use of a few basic extensions to the C language, and several general libraries, a model of a significant portion of an abstract MPI runtime has been constructed with moderate effort.

We presented a thorough evaluation of CIVL, including a comparison with other leading tools. We have shown that CIVL provides a number of features not provided by the other tools, including the ability to reason soundly over a wide range of nondeterministic behaviors of the MPI implementation. CIVL can be used to verify not only deadlock-freedom and correct use of MPI, but also the functional correctness of the computation performed by an MPI program. Other tools scale better than CIVL in some cases, but for realistic MPI applications, which often contain nondeterministic constructs, CIVL's performance is often competitive, thanks to the use of stateful model checking and precise partial order reduction techniques.

Future effort will focus on improving the performance of the CIVL verifier through parallelization and other optimizations, and supporting more MPI operations, such as nonblocking communication.

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

References

1. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of MPI programs. In: Malony, A.D., Nemirovsky, M., Midkiff, S.P. (eds.) Proceedings of the 27th international ACM conference on International

conference on supercomputing. pp. 455–456. ACM, ACM New York, NY, USA (2013), http://doi.acm.org/10.1145/2464996.2467286
2. Blaise Barney, L.L.N.L.: Lawrence Livermore National Laboratory Message-Passing Interface (MPI) exercise. https://computing.llnl.gov/tutorials/mpi/exercise.html (2016), accessed Feb. 8, 2015
3. Bronevetsky, G.: Communication-sensitive static dataflow for parallel message-passing applications. In: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 1–12. CGO '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/CGO.2009.32
4. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT press, Cambridge, MA, USA (1999)
5. Forejt, V., Kroening, D., Narayanaswamy, G., Sharma, S.: Precise predictive analysis for discovering communication deadlocks in MPI programs. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) Lecture Notes in Computer Science. Lecture Notes in Computer Science, vol. 8442, pp. 263–278. Springer, Cham (2014), http://dx.doi.org/10.1007/978-3-319-06410-9_19
6. Hilbrich, T., Protze, J., Schulz, M., de Supinski, B.R., Müller, M.S.: MPI Runtime Error Detection with MUST: Advances in Deadlock Detection. In: Hollingsworth, J.K. (ed.) International Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11–15, 2012. pp. 30:1–30:11. IEEE Computer Society Press, Los Alamitos, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2388996.2389037
7. International Organization for Standardization, 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)
8. Jordan, H., Pellegrini, S., Thoman, P., Kofler, K., Fahringer, T.: INSPIRE: The Insieme Parallel Intermediate Representation. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques. pp. 7–18. PACT '13, IEEE Press, Piscataway, NJ, USA (2013), http://dl.acm.org/citation.cfm?id=2523721.2523727
9. Jordan, H., Thoman, P., Fahringer, T.: A high-level IR transformation system. In: Euro-Par 2013: Parallel Processing Workshops, Aachen, Germany, August 26–27, 2013. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8374, pp. 647–656. Springer, Berlin, Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54420-0_63
10. López, H.A., Marques, E.R.B., Martins, F., Ng, N., Santos, C., Vasconcelos, V.T., Yoshida, N.: Protocol-based verification of message-passing parallel programs. In: Aldrich, J., Eugster, P. (eds.) Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015. pp. 280–298. ACM, New York, NY, USA (2015), http://doi.acm.org/10.1145/2814270.2814302
11. Marques, E.R.B., Martins, F., Vasconcelos, V.T., Ng, N., Martins, N.: Towards deductive verification of MPI programs against session types. In: Yoshida, N., Vanderbauwhede, W. (eds.) Proceedings 5th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, Rome, Italy, 23rd March 2013. Electronic Proceedings in Theoretical Computer Science, vol. 137, pp. 103–113. Open Publishing Association (2013), http://dx.doi.org/10.4204/EPTCS.137

12. Matlin, O.S., Lusk, E., McCune, W.: SPINning parallel systems software. In: Bosnacki, D., Leue, S. (eds.) Model Checking of Software: 9th Intl. SPIN Workshop. LNCS, vol. 2318, pp. 213–220. Springer, Berlin, Heidelberg (2002)
13. Message-Passing Interface Forum: MPI: A Message-Passing Interface standard, version 3.1. http://www.mpi-forum.org/docs/docs.html (Jun 2015)
14. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses formal methods. Commun. ACM 58(4), 66–73 (Mar 2015), http://doi.acm.org/10.1145/2699417
15. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs i. Acta Inf. 6, 319–340 (1976)
16. Santos, C., Martins, F., Vasconcelos, V.T.: Deductive verification of parallel programs using Why3. In: Knight, S., Lanese, I., Lluch-Lafuente, A., Vieira, H.T. (eds.) Proceedings 8th Interaction and Concurrency Experience, ICE 2015, Grenoble, France, 4-5th June 2015. EPTCS, vol. 189, pp. 128–142. ACM, New York, USA (2015), http://dx.doi.org/10.4204/EPTCS.189.11
17. Shires, D.R., Pollock, L.L., Sprenkle, S.: Program flow graph construction for static analysis of MPI programs. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 – July 1, 1999, Las Vegas, Nevada, USA. pp. 1847–1853. CSREA Press (1999)
18. Siegel, S.F.: Model Checking Nonblocking MPI Programs. In: Cook, B., Podelski, A. (eds.) Verification, Model Checking, and Abstract Interpretation: 8th International Conference, VMCAI 2007, Nice, France, January 14–16, 2007, Proceedings. LNCS, vol. 4349, pp. 44–58. Springer, Berlin, Heidelberg (2007)
19. Siegel, S.F., Avrunin, G.S.: Verification of MPI-based software for scientific computation. In: Graf, S., Mounier, L. (eds.) Model Checking Software: 11th International SPIN Workshop, Barcelona, Spain, April 1–3, 2004, Proceedings. LNCS, vol. 2989, pp. 286–303. Springer, Berlin, Heidelberg (2004)
20. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The Concurrency Intermediate Verification Language. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 61:1–61:12. SC '15, ACM, New York (2015), http://doi.acm.org/10.1145/2807591.2807635
21. Siegel, S.F., Zirkel, T.K.: FEVS: A Functional Equivalence Verification Suite for high performance scientific computing. Mathematics in Computer Science 5(4), 427–435 (2011)
22. Siegel, S.F., Zirkel, T.K.: TASS: The Toolkit for Accurate Scientific Software. Mathematics in Computer Science 5(4), 395–426 (2011)
23. Strout, M.M., Kreaseck, B., Hovland, P.D.: Data-flow analysis for MPI programs. In: Proceedings of the 2006 International Conference on Parallel Processing. pp. 175–184. ICPP '06, IEEE Computer Society, Washington, DC, USA (2006), http://dx.doi.org/10.1109/ICPP.2006.32
24. University of Utah, F.V.G.: Isp test results. http://formalverification.cs.utah.edu/ISP_Tests/ (2008), accessed May. 30, 2016
25. Vakkalanka, S., Gopalakrishnan, G., Kirby, R.M.: Dynamic verification of MPI programs with reductions in presence of split operations and relaxed orderings. In: Gupta, A., Malik, S. (eds.) Lecture Notes in Computer Science. Lecture Notes in Computer Science, vol. 5123, pp. 66–79. Springer, Berlin, Heidelberg (2008)
26. Vo, A., Aananthakrishnan, S., Gopalakrishnan, G., de Supinski, B.R., Schulz, M., Bronevetsky, G.: A scalable and distributed dynamic formal verifier for MPI programs. In: International Conference on High Performance Computing Networking,

Storage and Analysis, SC '10, New Orleans, LA, USA, November 13–19, 2010. pp. 1–10. IEEE/ACM, IEEE Computer Society Washington, DC, USA (2010), http://dx.doi.org/10.1109/SC.2010.7

A Complete List of Supported MPI Primitives

```
// types

typedef enum $operation MPI_Op;

typedef enum{
  MPI_CHAR,
  MPI_CHARACTER,
  MPI_SIGNED_CHAR,
  ...
}MPI_Datatype;

typedef struct MPI_Status{
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
  int size;
}MPI_Status;

typedef struct MPI_Comm MPI_Comm;

typedef struct MPI_Request * MPI_Request;

// functions

int MPI_Init(void );

int MPI_Finalize(void ) ;

double MPI_Wtime(void);

int MPI_Comm_size(MPI_Comm, int *) ;

int MPI_Comm_rank(MPI_Comm, int *) ;

int MPI_Send(const void *, int, MPI_Datatype, int, int, MPI_Comm);

int MPI_Recv(void *, int, MPI_Datatype, int, int, MPI_Comm,
     MPI_Status *) ;

int MPI_Get_count(MPI_Status *, MPI_Datatype, int *) ;

int MPI_Sendrecv(void *, int,
     MPI_Datatype,  int, int, void *, int, MPI_Datatype, int, int,
     MPI_Comm, MPI_Status *) ;

int MPI_Bcast(void *, int, MPI_Datatype, int, MPI_Comm) ;

int MPI_Reduce(const void *, void *, int, MPI_Datatype, MPI_Op, int,
     MPI_Comm) ;

int MPI_Allreduce(const void *, void *, int, MPI_Datatype, MPI_Op,
     MPI_Comm) ;

int MPI_Barrier(MPI_Comm);

int MPI_Gather(const void *,  int, MPI_Datatype, void *, int,
```

```
57      MPI_Datatype, int, MPI_Comm);
58
59  int MPI_Scatter(const void *, int, MPI_Datatype, void *, int,
60      MPI_Datatype, int, MPI_Comm);
61
62  int MPI_Gatherv(const void *, int, MPI_Datatype, void *, const int[],
63      const int[], MPI_Datatype, int, MPI_Comm);
64
65  int MPI_Scatterv(const void *, const int[], constint[], MPI_Datatype,
66      void *, int, MPI_Datatype, int, MPI_Comm);
67
68  int MPI_Allgather(const void *, int, MPI_Datatype, void *, int,
69      MPI_Datatype, MPI_Comm);
70
71  int MPI_Reducescatter(const void *, void *, constint[], MPI_Datatype,
72      MPI_Op, MPI_Comm) ;
73
74  int MPI_Alltoall(const void *, int, MPI_Datatype, void *, int,
75      MPI_Datatype, MPI_Comm) ;
76
77  int MPI_Alltoallv(const void *, const int[], const int[], MPI_Datatype,
78      void *, const int[], const int[], MPI_Datatype, MPI_Comm) ;
79
80  int MPI_Alltoallw(const void *, const int[], const int[],
81      const MPI_Datatype[], void *, constint[], const int[],
82      const MPI_Datatype[], MPI_Comm) ;
83
84  int MPI_Comm_dup(MPI_Comm, MPI_Comm *) ;
85
86  int MPI_Comm_free(MPI_Comm *) ;
87
88  int MPI_Init_thread( int *, char ***, int, int * );
```

B Libraries

The following are excerpts from the CIVL-C header files. They show all function prototypes, type and constant definitions that are used in the process of translating a C/MPI program to CIVL-C.

B.1 `civlc.cvh`: fundamental CIVL-C types and functions

```
1  // types
2
3  typedef enum Operation {
4    _NO_OP,     // no operation
5    _MAX,       // maximum
6    _MIN,       // minimun
7    _SUM,       // sum
8    _PROD,      // product
9    _LAND,      // logical and
10   _BAND,      // bit-wise and
11   _LOR,       // logical or
12   _BOR,       // bit-wise or
13   _LXOR,      // logical exclusive or
14   _BXOR,      // bit-wise exclusive or
15   _MINLOC,    // min value and location
16   _MAXLOC,    // max value and location
17   _REPLACE    // replace
18 } $operation;
19
```

```
// system functions

/*@ depends_on \nothing; */
$system void $wait($proc p);

/*@ depends_on \access(procs); */
$system void $waitall($proc *procs,int numProcs);

/*@ depends_on \nothing; */
$system void $exit(void);

/*@ depends_on \nothing; */
$system void $assert(sizeof_Bool expr, ...);

/*@ depends_on \nothing; */
$system void $assume(sizeof_Bool expr);

/*@ depends_on \nothing; */
$system void $elaborate(int x);

/*@ depends_on \nothing; */
$system int $next_time_count(void);

/*@ depends_on \nothing; */
$system void * $malloc($scope s, int size);

/*@ depends_on \access(p); */
$system void $free(void * p);
```

B.2 `pointer.cvh`: generalized CIVL-C pointer utilities

```
/*@ depends_on \access(obj1, obj2, result); */
$system void $apply(void *obj1, $operation op, void *obj2, void *result);

/*@ depends_on \access(ptr, value); */
$system void $copy(void *ptr, void *value);

/*@ depends_on \nothing; */
$system void *$pointer_add(const void *ptr, int offset, int type_size);
```

B.3 `seq.cvh`: sequences

```
/*@ depends_on \access(array); */
$system int $seq_length(void *array);

/*@ depends_on \access(array, value); */
$system void $seq_init(void *array,  int count, void *value);

/*@ depends_on \access(array, values); */
$system void $seq_insert(void *array,  int index, void *values,
     int count);

/*@ depends_on \access(array, values); */
$system void $seq_remove(void *array,  int index, void *values,
     int count);

/*@ depends_on \access(array, values); */
$atomic_f void $seq_append(void *array, void *values, int count);
```

B.4 `bundle.cvh`: bundles

```
// types
```

```
2
3  typedef struct _bundle $bundle;
4
5  // system functions
6
7  /*@ depends_on \nothing; */
8  $system int$bundle_size($bundle b);
9
10 /*@ depends_on \access(ptr); */
11 $system $bundle $bundle_pack(void *ptr, int size);
12
13 /*@ depends_on \access(ptr); */
14 $system void $bundle_unpack($bundle bundle, void *ptr);
15
16 /*@ depends_on \access(buf);
17 $system void $bundle_unpack_apply($bundle data, void *buf,
18      int size, $operation op);
```

B.5 concurrency.cvh: basic concurrency utilities

```
1  // types
2
3  typedef struct _gbarrier * $gbarrier;
4  typedef struct _barrier * $barrier;
5  typedef struct _collator_entry $collator_entry;
6  typedef struct _gcollator * $gcollator;
7  typedef struct _collator * $collator;
8
9  // functions
10
11 /*@ depends_on \nothing; */
12 $atomic_f $gbarrier $gbarrier_create($scope scope, int size);
13
14 /*@ depends_on \access(gbarrier); */
15 $atomic_f void $gbarrier_destroy($gbarrier gbarrier);
16
17 /*@ depends_on \nothing; */
18 $atomic_f $barrier $barrier_create($scope scope,
19      $gbarrier gbarrier, int place);
20
21 /*@ depends_on \access(barrier); */
22 $atomic_f void $barrier_destroy($barrier barrier);
23
24 void $barrier_call($barrier barrier);
25
26 /*@ depends_on \nothing; */
27 $atomic_f $gcollator $gcollator_create($scope scope);
28
29 /*@ depends_on \access(gcollator); */
30 $atomic_f int $gcollator_destroy($gcollator gcollator);
31
32 /*@ depends_on \nothing; */
33 $atomic_f $collator $collator_create($scope scope, $gcollator gcollator);
34
35 /*@ depends_on \access(collator); */
36 $atomic_f void $collator_destroy($collator collator);
37
38 /*@ depends_on \nothing; */
39 $system $bundle $collator_check($collator collator,
40      int place, int nprocs, $bundle entries);
```

B.6 comm.cvh: basic message-passing communication primitives

```
// constants

#define $COMM_ANY_SOURCE -1
#define $COMM_ANY_TAG -2

// types

typedef struct _message $message;
typedef struct _queue $queue;
typedef struct _gcomm * $gcomm;
typedef struct _comm * $comm;

// functions

/*@ depends_on \access(data); */
$atomic_f $message $message_pack(int source, int dest, int tag,
     void *data, int size);

/*@ depends_on \nothing; */
$atomic_f int $message_source($message message);

/*@ depends_on \nothing; */
$atomic_f int $message_tag($message message);

/*@ depends_on \nothing; */
$atomic_f int $message_dest($message message);

/*@ depends_on \nothing; */
$atomic_f int $message_size($message message);

/*@ depends_on \access(buf); */
$atomic_f void $message_unpack($message message, void *buf, int size);

/*@ depends_on \nothing; */
$atomic_f $gcomm $gcomm_create($scope scope, int size);

/*@ depends_on \access(junkMsgs, gcomm); */
$atomic_f int $gcomm_destroy($gcomm gcomm, void *junkMsgs);

/*@ depends_on \access(comm, newcomm);
$system void $gcomm_dup($comm comm, $comm newcomm);

/*@ depends_on \nothing; */
$atomic_f $comm $comm_create($scope scope, $gcomm gcomm, int place);

/*@ depends_on \access(comm); */
$atomic_f void $comm_destroy($comm comm);

/*@ depends_on \nothing; */
$atomic_f int $comm_size($comm comm);

/*@ depends_on \nothing; */
$atomic_f int $comm_place($comm comm);

/*@ depends_on \access(comm); */
$system void $comm_enqueue($comm comm, $message message);

/*@ depends_on \access(comm); */
$system $message $comm_seek($comm comm, int source, int tag);

/*@ depends_on \access(comm); */
$system sizeof_Bool $comm_probe($comm comm, int source, int tag);
```

```
63
64 /*@ depends_on \access(comm);
65   @ executes_when $comm_probe(comm, source, tag); */
66 $system $message $comm_dequeue($comm comm, int source, int tag);
```

B.7 civl-mpi.cvh: CIVL MPI support library

```
1  struct _mpi_gcomm $mpi_gcomm;
2
3  typedef enum _mpi_state {
4    _MPI_UNINIT, _MPI_INIT, _MPI_FINALIZED
5  } $mpi_state;
6
7  typedef enum{
8    MPI_CHAR,
9    MPI_CHARACTER,
10   MPI_SIGNED_CHAR,
11   MPI_UNSIGNED_CHAR,
12   MPI_BYTE,
13   MPI_WCHAR,
14   MPI_SHORT,
15   MPI_UNSIGNED_SHORT,
16   MPI_INT,
17   MPI_INT16_T,
18   MPI_INT32_T,
19   MPI_INT64_T,
20   MPI_INT8_T,
21   MPI_INTEGER,
22   MPI_INTEGER1,
23   MPI_INTEGER16,
24   MPI_INTEGER2,
25   MPI_INTEGER4,
26   MPI_INTEGER8,
27   MPI_UNSIGNED,
28   MPI_LONG,
29   MPI_UNSIGNED_LONG,
30   MPI_FLOAT,
31   MPI_DOUBLE,
32   MPI_LONG_DOUBLE,
33   MPI_LONG_LONG_INT,
34   MPI_UNSIGNED_LONG_LONG,
35   MPI_LONG_LONG,
36   MPI_PACKED,
37   MPI_LB,
38   MPI_UB,
39   MPI_UINT16_T,
40   MPI_UINT32_T,
41   MPI_UINT64_T,
42   MPI_UINT8_T,
43   MPI_FLOAT_INT,
44   MPI_DOUBLE_INT,
45   MPI_LONG_INT,
46   MPI_SHORT_INT,
47   MPI_2INT,
48   MPI_LONG_DOUBLE_INT,
49   MPI_AINT,
50   MPI_OFFSET,
51   MPI_2DOUBLE_PRECISION,
52   MPI_2INTEGER,
53   MPI_2REAL,
54   MPI_C_BOOL,
55   MPI_C_COMPLEX,
56   MPI_C_DOUBLE_COMPLEX,
```

```
  MPI_C_FLOAT_COMPLEX,
  MPI_C_LONG_DOUBLE_COMPLEX,
  MPI_COMPLEX,
  MPI_COMPLEX16,
  MPI_COMPLEX32,
  MPI_COMPLEX4,
  MPI_COMPLEX8,
  MPI_REAL,
  MPI_REAL16,
  MPI_REAL2,
  MPI_REAL4,
  MPI_REAL8
} MPI_Datatype;

int sizeofDatatype(MPI_Datatype);

$abstract double$mpi_time(int i);

$mpi_gcomm $mpi_gcomm_create($scope, int);

void $mpi_gcomm_destroy($mpi_gcomm);

MPI_Comm $mpi_comm_create($scope, $mpi_gcomm, int);

void $mpi_comm_destroy(MPI_Comm, $mpi_state);

int $mpi_send(const void *, int, MPI_Datatype, int, int, MPI_Comm);

int $mpi_recv(void *, int, MPI_Datatype, int, int, MPI_Comm,
     int,  MPI_Status *);

int $mpi_sendrecv(const void *, int, MPI_Datatype, int, int, void *, int,
     MPI_Datatype, int, int,  MPI_Comm, MPI_Status *);

int $mpi_collective_send(const void *, int, MPI_Datatype, int, int,
     MPI_Comm);

int $mpi_collective_recv(void *, int, MPI_Datatype, int, int, MPI_Comm,
     MPI_Status *, char *);

int $mpi_bcast(void *, int, MPI_Datatype, int, int, MPI_Comm,  char *);

int $mpi_reduce(const void *, void *, int,  MPI_Datatype, MPI_Op,
     int, int, MPI_Comm, char *);

int $mpi_gather(const void *, int, MPI_Datatype, void *, int,
     MPI_Datatype, int, int, MPI_Comm, char *);

int $mpi_gatherv(const void *, int, MPI_Datatype, void*,
     const int[], const int[],  MPI_Datatype, int, int,  MPI_Comm,
     char *);

int $mpi_scatter(const void *, int, MPI_Datatype,  void*, int,
     MPI_Datatype, int,  int, MPI_Comm, char *);

int $mpi_scatterv(const void *, const int[], const int[], MPI_Datatype,
     void *, int, MPI_Datatype, int, int,  MPI_Comm, char*);

void* $mpi_pointer_add(void*, int, MPI_Datatype);

$system int $mpi_new_gcomm($scope, $mpi_gcomm);

$system $mpi_gcomm $mpi_get_gcomm($scope, int);
```

```
int $mpi_comm_dup($scope, MPI_Comm, MPI_Comm*, char*);

int $mpi_comm_free(MPI_Comm*, $mpi_state);

$system $scope $mpi_root_scope($comm);

$system $scope $mpi_proc_scope($comm);

$system void $mpi_assert_consistent_basetype(void*, int, MPI_Datatype);

$bundle $mpi_create_routine_entry(int, int, int, int, int*);

void $mpi_diff_coroutine_entries($bundle, $bundle, int);
```

C Excerpts of Implementation of Libraries

C.1 seq.cvl: implementing seq.cvh

```
/*@ depends_on \access(array, values); */
$atomic_f void $seq_append(void * array, void * values, int count){
  int length = $seq_length(array);

  $seq_insert(array, length, values, count);
}
```

C.2 concurrency.cvl: implementing concurrency.cvh

```
/*@ depends_on \access(barrier); */
$system void $barrier_enter($barrier barrier);

/*@ depends_on \access(barrier); */
$system void $barrier_exit($barrier barrier);

void $barrier_call($barrier barrier) {
  $barrier_enter(barrier);
  $barrier_exit(barrier);
}
```

C.3 comm.cvl: implementing concurrency.cvh

```
struct _queue {
  int length;
  $message messages[];
};

struct _gcomm {
  int nprocs; // number of processes
  $proc procs[]; // process references
  sizeof_Bool isInit[]; // if the local comm has been initiated
  $queue buf[][]; // message buffers
};

struct _comm {
  int place;
  $gcomm gcomm;
};

/*@ depends_on \access(data); */
$atomic_f $message $message_pack(int source, int dest, int tag,
```

```
20      void * data, int size){
21    $message result;
22
23    result.source = source;
24    result.dest = dest;
25    result.tag = tag;
26    result.data = $bundle_pack(data, size);
27    result.size = size;
28    return result;
29  }
30
31  /*@ depends_on \access(buf); */
32  $atomic_f void $message_unpack($message message, void * buf, int size){
33    $bundle_unpack(message.data, buf);
34    $assert(message.size <= size, "Message of size %d exceeds the
35         specified size %d.", message.size, size);
36  }
37
38  /*@ depends_on \nothing; */
39  $atomic_f $gcomm $gcomm_create($scope scope, int size){
40    $gcomm gcomm=($gcomm)$malloc(scope, sizeof(struct _gcomm));
41    $queue empty;
42
43    empty.length=0;
44    $seq_init(&empty.messages, 0, NULL);
45    gcomm->nprocs=size;
46    gcomm->procs=($proc[size])$lambda(int i) $proc_null;
47    gcomm->isInit=(sizeof_Bool[size])$lambda(int i) $false;
48    gcomm->buf=($queue[size][size])$lambda(int i,j) empty;
49    return gcomm;
50  }
51
52  /*@ depends_on \nothing; */
53  $atomic_f $comm $comm_create($scope scope, $gcomm gcomm, int place){
54    $assert(!gcomm->isInit[place], "the place %d is already occupied in
55       the global communicator!", place);
56
57    $comm comm=($comm)$malloc(scope, sizeof(struct _comm));
58
59    gcomm->procs[place]=$self;
60    gcomm->isInit[place]=$true;
61    comm->gcomm=gcomm;
62    comm->place=place;
63    return comm;
64  }
65
66  /*@ depends_on \access(comm); */
67  $atomic_f int $comm_place($comm comm){
68    return comm->place;
69  }
```

C.4 `civl-mpi.cvl`: implementing `civl-mpi.cvh`

```
1   typedef struct MPI_Comm{
2     $comm p2p; // point-to-point communication
3     $comm col; // collective communication
4     $collator collator;
5     $barrier barrier;
6     int gcommIndex; //the index of the corresponding global communicator
7   }MPI_Comm;
8
9   struct $mpi_gcomm{
10    $gcomm p2p; // point-to-point communication
```

```
11    $gcomm col; // collective communication
12    $gcollator gcollator;
13    $gbarrier gbarrier;
14  };
15
16  int $mpi_send(void * buf, int count, MPI_Datatype datatype, int dest,
17       int tag, MPI_Comm comm) {
18    if (dest >= 0) {
19      int size = count*sizeofDatatype(datatype);
20      int place = $comm_place(comm.p2p);
21      $message out = $message_pack(place, dest, tag, buf, size);
22
23      $comm_enqueue(comm.p2p, out);
24    }
25    return 0;
26  }
27
28  int $mpi_recv(void * buf, int count, MPI_Datatype datatype, int source,
29       int tag, MPI_Comm comm, MPI_Status * status) {
30    if (source >= 0 || source == MPI_ANY_SOURCE) {
31      $message in;
32      int place = $comm_place(comm.p2p);
33      int deterministicTag;
34
35      $assert(tag == -2 || tag >= 0,
36           "Illegal MPI message receive tag %d.\n", tag);
37      deterministicTag = tag < 0 ? -2 : tag;
38      $elaborate(source);
39      in = $comm_dequeue(comm.p2p, source, deterministicTag);
40
41      int size = count*sizeofDatatype(datatype);
42
43      $message_unpack(in, buf, size);
44      if (status != MPI_STATUS_IGNORE) {
45        status->size = $message_size(in);
46        status->MPI_SOURCE = $message_source(in);
47        status->MPI_TAG = $message_tag(in);
48        status->MPI_ERROR = 0;
49      }
50    }
51    return 0;
52  }
53
54  int $mpi_reduce(const void * sendbuf, void * recvbuf, int count,
55       MPI_Datatype datatype, MPI_Op op, int root, int tag,  MPI_Comm comm,
56       char * routName) {
57    int rank;
58
59    rank = $comm_place(comm.col);
60    if (rank != root)
61      $mpi_collective_send(sendbuf, count, datatype, root, tag, comm);
62    else {
63      int nprocs = $comm_size(comm.col);
64      int size;
65
66      size = count * sizeofDatatype(datatype);
67      memcpy(recvbuf, sendbuf, size);
68      $for(int i = 0; i<nprocs; i++) {
69        if(i != root){
70          int colTag;
71          $message in = $comm_dequeue(comm.col, i, MPI_ANY_TAG);
72
73          colTag = $message_tag(in);
74          $assert(colTag == tag , "Collective routine %s receives a "
```

```
75              "message with a mismatched tag\n", routName);
76          $bundle_unpack_apply(in.data, recvbuf, count, op);
77          $assert(in.size <= size , "Message of size %d exceeds
78              the specified size %d.", in.size, size);
79        }
80      }
81    }
82    return 0;
83  }
```

C.5 `mpi.cvl`: implementing `mpi.h`

```
1  int MPI_Send(const void * buf, int count, MPI_Datatype datatype, int dest,
2       int tag, MPI_Comm comm){
3    $assert(_mpi_state == _MPI_INIT, "MPI_Send() cannot be invoked "
4         "without MPI_Init() being called before.\n");
5    $mpi_check_buffer(buf, count, datatype);
6    return $mpi_send(buf, count, datatype, dest, tag, comm);
7  }
8
9  int MPI_Recv(void * buf, int count, MPI_Datatype datatype, int source,
10      int tag, MPI_Comm comm, MPI_Status *  status) {
11   $assert(_mpi_state == _MPI_INIT, "MPI_Recv() cannot be invoked "
12        "without MPI_Init() being called before.\n");
13   return $mpi_recv(buf, count, datatype, source, tag, comm, status);
14 }
15
16 int MPI_Reduce(const void * sendbuf, void * recvbuf, int count,
17      MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) {
18   int place = $comm_place(comm.col);
19   int nprocs = $comm_size(comm.col);
20   int datatypes[1] = {(int)datatype};
21   // MPI library defined collective operation checking entries:
22   $bundle checkerEntry; // the checking entry of this call
23   $bundle specEntry; // a recorded entry as specification
24
25   checkerEntry = $mpi_create_routine_entry(_REDUCE_TAG, root, (int)op,
26        1, datatypes);
27   specEntry = $collator_check(comm.collator, place, nprocs, checkerEntry);
28   $mpi_diff_coroutine_entries(specEntry, checkerEntry, place);
29   $mpi_check_buffer(sendbuf, count, datatype);
30   $mpi_reduce(sendbuf, recvbuf, count, datatype, op, root, _REDUCE_TAG,
31        comm, "MPI_Reduce()");
32   return 0;
33 }
```

D Experiments and Examples

D.1 Results of the experiment suite

Results of verifying C/MPI programs using CIVL, MPI-Spin, and TASS. Presence of superscript c indicates a comparison with a sequential version was performed, otherwise single-program verification was performed. Tools: ta=TASS, ms=MPI-Spin, otherwise CIVL was used. Result: + (all properties hold), or - (violation found).

program	R	states	trans.	time	scale
diffusion1d.c^{c} [21]	+	285961	281860	45	NP∈[1,3], NSTEPS,NX∈[1,5], WSTEP∈[1,NSTEPS]
diffusion1d.c	+	156479	154517	34	NP∈[1,3], NSTEPS,NX∈[1,5], WSTEP∈[1,NSTEPS]
diffusion2d.c^{c} [21]	+	779098	768852	208	NPX=NPY=2, NP=NPX*NPY, NSTEPS, NX, NY∈[1,5]
diffusion2d.c	+	699656	690734	201	NPX=NPY=2, NP=NPX*NPY, NSTEPS, NX, NY∈[1,5]
diffusion.prom^{ms} [18]	+	26041	41637	1	NPX=NX=3,NPY=NY=1,NSTEPS=2
matmat_mw.c^{c} [21]	+	119872	118917	20	NP∈[2,4], N,L,M∈[1,3]
matmat_mw.c	+	82772	81847	19	NP∈[2,4], N,L,M∈[1,3]
matmat_mw.c^{c}	+	673808	670230	357	NP=5, N=L=M=8
matmat_mw.prom^{ms,c} [18]	+	153920	360875	7	NP=5, N=L=M=8
mpi_pi_send.c [2]	+	135895	134765	30	1<=ROUNDS, DARTS, NP<=2
mpi_prime.c [2]	+	46563	46810	36	1<=NP<=4, 10<=PRIMES<=15
mpithread_both.c [20]	+	109868	127027	28	NP=2, MAXTHRDS=2, VECLEN=5
gauss_elim.c [20]	+	349869	348016	97	NP=ROW=COL=3
gauss_elim.c^{c} [21]	+	287641	285584	93	NP=ROW=COL=3
gauss_elim.prom^{ms,c} [18]	+	1947	2526	<1	NP=ROW=COL=3
wave1d.c^{c} [20]	+	143385	141144	31	1<=NP<=4, 1<=NSTEPS, NX<=5
wave1d.c	+	127159	125234	30	1<=NP<=4, 1<=NSTEPS, NX<=5
wave1dBad.c [20]	-	505	504	3	1<=NP<=4, 1<=NSTEPS, NX<=5
c_ex01.c [**?**]	+	956	952	4	NP=6
c_ex04.c [**?**]	+	1710	1703	4	NP=6
c_ex05.c [**?**]	+	3122	3110	4	NP=6
c_ex06.c [**?**]	+	2768	2761	4	NP=6
c_ex07.c [**?**]	+	5320	5288	6	NP=6
c_ex08_inconsist_col.c	-	391	389	4	NP=6
c_ex13.c [**?**]	+	3249	3237	5	NP=6
any_src-can-deadlock.c [24]	-	2125	2120	3	NP=6
any_src-can-deadlock.c^{ta} [24]	-	76054	76312	5	NP=6
any_src-deadlock.c [24]	+	1616	1612	3	NP=6
basic-deadlock.c [24]	-	899	895	3	NP=6
basic-deadlock-comm_dup.c [24]	-	1782	1773	4	NP=6
bcast-deadlock.c [24]	-	257	255	3	NP=6
collective-misorder.c [24]	-	1026	1024	3	NP=6
comm-dup-no-error.c [24]	+	20523	20421	21	NP=6
comm-dup-no-free.c [24]	-	1569	1563	3	NP=6
complex-deadlock.c [24]	-	959	954	3	NP=6
complex-deadlock.c^{ta} [24]	-	230	229	<1	NP=6
deadlock-config.c [24]	-	596	594	2	NP=6
sendrecv-deadlock.c [24]	-	943	939	3	NP=6
sendrecv-deadlock.c^{ta} [24]	-	1768	1767	<1	NP=6
send-recv-ok.c [24]	+	1616	1612	3	NP=6
no-error-any_src.c [24]	+	5670	5681	4	NP=6
no-error.c [24]	+	1320	1312	3	NP=6

D.2 dybuf.c

```
1 int nprocs, rank, tag=0; MPI_Comm comm = MPI_COMM_WORLD;
2 MPI_Status *stat = MPI_STATUS_IGNORE;
3 /* If all communication is synchronous, rank 0 must return 0.
4  * If communication can buffer, rank 0 could return 0 or 1. */
5 int f() {
6   int buf;
7   if (rank == 0) {
8     MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, stat);
9     MPI_Recv(&buf, 1, MPI_INT, MPI_ANY_SOURCE, tag, comm, stat);
10    return buf;
11  } else if (rank == 1) }
12    MPI_Recv(NULL, 0, MPI_INT, 2, tag, comm, stat);
13    buf = 0;
14    MPI_Send(&buf, 1, MPI_INT, 0, tag, comm);
15  } else if (rank == 2) {
16    buf = 1;
17    MPI_Send(&buf, 1, MPI_INT, 0, tag, comm);
18    MPI_Send(NULL, 0, MPI_INT, 1, tag, comm);
19  }
20  return 0;
21 }
22
23 /* Deadlocks iff all communication is synchronous. */
24 void g() {
25   if (rank == 0) {
26     MPI_Send(NULL, 0, MPI_INT, 1, tag, comm);
27     MPI_Recv(NULL, 0, MPI_INT, 1, tag, comm, stat);
28   } else if (rank == 1) {
29     MPI_Send(NULL, 0, MPI_INT, 0, tag, comm);
30    MPI_Recv(NULL, 0, MPI_INT, 0, tag, comm, stat);
31   }
32 }
33
34 /* May or may not deadlock */
35 int main(int argc, char * argv[]) {
36   MPI_Init(&argc, &argv);
37   MPI_Comm_size(comm, &nprocs);
38   MPI_Comm_rank(comm, &rank);
39   assert(nprocs >= 3);
40   intx = f();
41   MPI_Barrier(comm);
42   MPI_Bcast(&x, 1, MPI_INT, 0, comm);
43   if (x) g();
44   MPI_Finalize();
45 }
```