

CIVL

The Concurrency Intermediate Verification Language
Reference Manual
v0.10

Matthew B. Dwyer	Ganesh Gopalakrishnan	Ziqing Luo
Zvonimir Rakamaric	Stephen F. Siegel	Manchun Zheng
Timothy K. Zirkel		

March 28, 2014

Contents

I	Introduction	4
1	Acknowledgement	5
2	What is CIVL?	6
3	Installation and Quick Start	7
4	Examples	8
4.1	Dining Philosophers	8
4.2	A Multithreaded MPI Example	9
II	Language	14
5	Overview of CIVL-C	15
5.1	Main Concepts	15
5.2	Example Illustrating Scopes and Processes	15
5.3	Structure of a CIVL-C program	17
6	Sequential Elements	19
6.1	Types	19
6.1.1	Standard types inherited from C	19
6.1.2	The bundle type: <code>\$bundle</code>	19
6.1.3	The <code>\$scope</code> type	19
6.2	Expressions	20
6.2.1	Expressions inherited from C	20
6.2.2	Scope expressions	21
6.2.2.1	Checking if a dyscope is defined: <code>\$scope_defined</code>	21
6.2.2.2	The constant <code>\$here</code>	21
6.2.2.3	The constant <code>\$root</code>	21
6.2.2.4	Scope relational operators	21
6.2.2.5	Scope parent function <code>\$scope_parent</code>	22
6.2.2.6	Lowest Common Ancestor: <code>+</code>	22
6.2.2.7	The <code>\$scopeof</code> expression	22
6.3	Statements	23
6.4	Guards and nondeterminism	23
6.4.1	Guarded commands: <code>\$when</code>	23
6.4.2	Nondeterministic selection statement: <code>\$choose</code>	24

6.4.3	Nondeterministic choice of integer: <code>\$choose_int</code>	25
7	Concurrency	26
7.1	Process creation and management	26
7.1.1	The process type: <code>\$proc</code>	26
7.1.2	Checking if a process is defined: <code>\$proc_defined</code>	26
7.1.3	Obtaining the null process reference: <code>\$proc_null</code>	26
7.1.4	The <i>self</i> process constant: <code>\$self</code>	26
7.1.5	Spawning a new process: <code>\$spawn</code>	27
7.1.6	Waiting for another process to terminate: <code>\$wait</code>	27
7.1.7	Terminating a process immediately: <code>\$exit</code>	27
7.2	Atomicity	27
7.2.1	Atom blocks: <code>\$atom</code>	27
7.2.2	Atomic blocks: <code>\$atomic</code>	28
7.3	Message-Passing	29
7.3.1	Messages: <code>\$message</code>	29
7.3.2	Communicators: <code>\$gcomm</code> and <code>\$comm</code>	29
8	Specification	33
8.1	Overview	33
8.2	Input-output signature	33
8.2.1	Input type qualifier: <code>\$input</code>	33
8.2.2	Output type qualifier: <code>\$output</code>	34
8.3	Assertions and assumptions	34
8.3.1	Assertions: <code>\$assert</code>	34
8.3.2	Assume statements: <code>\$assume</code>	34
8.4	Formulas	35
8.4.1	Implication: <code>=></code>	35
8.4.2	Universal quantifier: <code>\$forall</code>	35
8.4.3	Existential quantifier: <code>\$exists</code>	35
8.5	Contracts	36
8.5.1	Procedure contracts: <code>\$requires</code> and <code>\$ensures</code>	36
8.5.2	Loop invariants: <code>\$invariant</code>	36
8.6	Concurrency specification	36
8.6.1	Remote expressions: <code>e@x</code>	36
8.6.2	Collective expressions: <code>\$collective</code>	37
9	Pointers and Heaps	38
9.1	Memory functions: <code>memcpy</code>	38
9.2	Heaps, <code>\$malloc</code> and <code>\$free</code>	38
10	Libraries	39
III	Semantics	40
11	CIVL Model Syntax	41
11.1	Notation and terminology	41

11.2	Definition of Context	41
11.3	Lexical scopes	42
11.4	Functions	44
11.5	Statements	45
11.6	Remarks	46
11.7	Transition system representation of functions	47
12	CIVL Model Semantics	48
12.1	State	48
12.2	Jump protocol	49
12.3	Initial State	51
12.4	Transitions	51
12.5	Calls and Spawns	51
12.6	Garbage collection	52
IV	Tools	53
13	Tool Overview	54
13.1	Symbolic execution	54
13.2	Commands	54
13.3	Options	55
13.4	Errors	56
14	Interpreting the Output	58
14.1	Transitions	58
14.2	States	59
14.3	Property Violations	61
14.4	Statistics	62
15	Emacs mode	63
	Bibliography	64

Part I

Introduction

Chapter 1

Acknowledgement

The CIVL project is funded by the U.S. National Science Foundation under awards CCF-1346769 and CCF-1346756.

Chapter 2

What is CIVL?

CIVL stands for *Concurrency Intermediate Verification Language*. The *CIVL platform* encompasses:

1. the programming language **CIVL-C**, a dialect of C with additional primitives supporting concurrency, specification, and modeling;
2. verification and analysis tools, including a symbolic execution-based model checker for checking various properties of, or finding defects in, CIVL-C programs; and
3. tools that translate from many commonly used languages/APIs to CIVL-C.

The CIVL-C language is primarily intended to be an intermediate representation for verification. A C program using MPI [2], CUDA [3], OpenMP [4], OpenCL [1], or another API (or even some combination of APIs), will be automatically translated into CIVL-C and then verified. The advantages of such a framework are clear: the developer of a new verification technique could implement it for CIVL-C and then immediately see its impact across a broad range of concurrent programs. Likewise, when a new concurrency API is introduced, one only needs to implement a translator from it to CIVL-C in order to reap the benefits of all the verification tools in the platform. Programmers would have a valuable verification and debugging tool, while API designers could use CIVL as a “sandbox” to investigate possible API modifications, additions, and interactions.

This manual covers all aspects of the CIVL framework, and is organized in parts as follows:

1. this introduction, including “quick start” instructions for downloading and installing CIVL and several examples;
2. a complete description of the CIVL-C language;
3. a formal semantics for the language; and
4. a description of the tools in the framework.

Chapter 3

Installation and Quick Start

This chapter gives instructions for downloading and installing CIVL, and running the verification tool on an example.

Note. The following instructions explain how to install CIVL under the directory `/opt`, which is the standard installation point. If you do not wish to install CIVL there (e.g., because you don't want to use `sudo`, or you want to install it in your home directory), you can just modify the instructions by replacing `/opt` with another directory. You will just need to edit the script file `civl` appropriately, replacing the default paths with your chosen paths.

1. Download and unpack the appropriate pre-compiled library of CIVL dependencies from http://vsl.cis.udel.edu/tools/vsl_depend/. There are versions for Darwin (OS X), 32-bit Linux, and 64-bit Linux. After unpacking, you should have a directory named `vsl`.
2. If you do not already have a directory `/opt`, create it; use `sudo` as needed (`sudo mkdir /opt`).
3. Move the directory `vsl` into `/opt` (`mv vsl /opt`).
4. Download and unpack the latest stable release of CIVL from <http://vsl.cis.udel.edu/civl>. Again there are versions for Darwin, 32-bit Linux, and 64-bit Linux.
5. The resulting directory should be named `CIVL-tag` for some string `tag` which identifies the version of CIVL you downloaded. Move this directory into `/opt` (`mv CIVL-tag /opt`).
6. There should now be an executable script at `/opt/CIVL-tag/bin/civl`. Move this script into your path, or create a symlink from somewhere in your path to it, or add the directory `/opt/CIVL-tag/bin` to your path.

From the command line, you should now be able to type “`civl help`” and see a help message describing the command line syntax.

Copy the file `/opt/CIVL-tag/examples/concurrency/locksBad.cvl` to your working directory. Look at the program: it is a simple 2-process program with two shared variables used as locks. The two processes try to obtain the locks in opposite order, which can lead to a deadlock if both processes obtain their first lock before either obtains the second. Type “`civl verify locksBad.cvl`”. You should see some output culminating in a message

The program MAY NOT be correct. See `CIVLREP/locksBad_log.txt`

Type “`civl replay locksBad.cvl`”. You should see a step-by-step account of how the program arrived at the deadlock.

Chapter 4

Examples

In this section we show a few simple CIVL-C programs which illustrate some of the pertinent features of the language. We also show the results of running some of the tools on them.

4.1 Dining Philosophers

Dijkstra’s well-known Dining Philosophers system can be encoded in CIVL-C as shown in Figure 4.1.

In this encoding, an upper bound B is placed on the number of philosophers n . When verifying this program, a concrete value will be specified for B . Hence the result of verification will apply to all n between 2 and B , inclusive.

Both B and n are declared as *input* variables using the type qualifier `$input`. An input variable may be initialized with any valid value of its type. In contrast, non-input variables declared in file scope will be initialized with a special *undefined* value; if such a variable is read before it is defined, an error will be reported. In addition, any input variable may have a concrete initial value specified on the command line. In this case, we will specify a concrete value for B on the command line but leave n unconstrained.

An `$assume` statement restricts the set of executions of the program to include only those traces in which the assumptions hold. In contrast with an `$assert` statement, CIVL does not check that the assumed expression holds, and will not generate an error message if it fails to hold. Thus an `$assume` statement allows the programmer to say to CIVL “assume that this is true,” while an `$assert` statement allows the programmer to say to CIVL “check that this is true.”

A `$when` statement encodes a *guarded command*. The `$when` statement includes a boolean expression called the *guard* and a statement body. The `$when` statement is enabled if and only if the *guard* evaluates to *true*, in which case the body may be executed. The first atomic statement in the body executes atomically with the evaluation of the guard, so it is guaranteed that the guard will hold when this initial sub-statement executes. Since assignment statements are atomic in CIVL, in this example the body of each `$when` statement executes atomically with the guard evaluation.

The `$spawn` statement is very similar to a function call. The main difference is that the function called is invoked in a new process which runs concurrently with the existing processes. The `$spawn` statement itself returns immediately.

The program may be verified for an upper bound of 5 by typing the following at the command line:

```
civil verify -inputB=5 diningBad.cvl
```

```

#include <civlc.h>

$input int B; // upper bound on number of philosophers
$input int n; // number of philosophers
$assume 2<=n && n<=B;
_Bool forks[n]; // Each fork will be on the table (0) or in a hand (1).

void dine(int id) {
    int left = id;
    int right = (id + 1) % n;

    while (1) {
        $when (forks[left] == 0) forks[left] = 1;
        $when (forks[right] == 0) forks[right] = 1;
        forks[right] = 0;
        forks[left] = 0;
    }
}

void main() {
    for (int i = 0; i < n; i++) forks[i] = 0;
    for (int i = 0; i < n; i++) $spawn dine(i);
}

```

Figure 4.1: diningBad.cvl: CIVL-C encoding of Dijkstra’s Dining Philosophers

The output indicates that a deadlock has been found and a counterexample has been produced and saved. We can examine the counterexample, but it is more helpful to work with a *minimal* counterexample, i.e., a deadlocking trace of minimal length. To find a minimal counterexample, we issue the command

```
civl verify -inputB=5 -min diningBad.cvl
```

The result of this command is shown in Figure 4.2. The output indicates that a minimal counterexample has length 19, i.e., involves 20 states and 19 transitions (the depth of 20 is one more than 19). It was the 26th and shortest trace found. It was deemed equivalent to the earlier traces and hence the earlier ones were discarded and only this one saved. We can replay the trace with the command

```
civl replay diningBad.cvl
```

The result of this command is shown in Figure 4.3. The output indicates that a deadlock has been found involving 2 philosophers. The trace has 15 transitions; after the initialization sequence, each philosopher picks up her left fork.

4.2 A Multithreaded MPI Example

Figure 4.4 is an example of a CIVL-C model of multithreaded MPI program. The program consists of two processes, each of which spawns two threads. All four threads issue message-passing operations.

This example illustrates some of the message-passing primitives provided in CIVL-C. A *global communicator* object is allocated in the root scope. The constant **\$here** has type **\$scope** and refers to the scope in which the expression occurs; in this case it is the root (i.e., file) scope. This global communicator is declared to have **NPROCS** *places*; these are points from which messages can be sent or received. The function **MPI_Process** is used to model an MPI process. Each instance will create its own *local communicator* object which specifies the global communicator and a place; this is the object that will be used to send or receive messages at that place.

Each process spawns two instances of function **Thread**. Each thread creates a message object from a buffer, specifying the source and destination places, tag, pointer to the beginning of the buffer, and the size of the buffer. The message is *enqueued* into the communication universe using the local communicator. Similarly, messages are dequeued by specifying the local communicator, source place, and tag.

The program has a subtle defect, which only manifests on very specific interleavings of the threads. This defect can be found using **civil verify**.

```

CIVL v0.10 of 2014-03-28 -- http://vsl.cis.udel.edu/civl
Error 0 encountered at depth 129:
...
Error 25 encountered at depth 16:
CIVL execution error (kind: DEADLOCK, certainty: PROVEABLE)
A deadlock is possible:
  Path condition: true
  Enabling predicate: false
ProcessState 0: terminated
ProcessState 1: at location 26, f0:21.30-42 "forks[right]"
  Enabling predicate: false
ProcessState 2: at location 26, f0:21.30-42 "forks[right]"
  Enabling predicate: false
at f0:21.30-42 "forks[right]".
State 664
| Path condition
| | true
| Dynamic scopes
| | dyscope 0 (parent=-1, static=0)
| | | reachers = {1,2}
| | | variables
| | | | __atomic_lock_var = process<-1>
| | | | B = 5
| | | | n = 2
| | | | forks = X_s0v4[0:=1, 1:=1]
...
| Process states
...
| | process 2
| | | atomicCount = 0
| | | call stack
| | | | Frame[function=dine, location=25, f0:21.30-42 "forks[right]", scope=3]
...
===== Stats =====
  validCalls      : 15327
  proverCalls     : 17
  memory (bytes)  : 18554880
  time (s)        : 2.17
  maxProcs        : 6
  statesInstantiated : 9264
  statesSaved      : 665
  statesSeen       : 1758
  statesMatched    : 1177
  steps           : 2993
  transitions      : 2934

```

The program MAY NOT be correct. See CIVLREP/diningBad_log.txt

Figure 4.2: Output from `civl verify -inputB=5 -min diningBad.cvl`

```

...
Transition 1: State 0, proc 0:
  0->1: B = 5 at f0:9.0-12 "$input int B";
  1->2: n = InitialValue(n) at f0:10.0-12 "$input int n";
  2->3: $assume ((2<=n)&&(n<=B)) at f0:11.0-20 "$assume 2<=n && n ... B";
  3->5: forks = InitialValue(forks) at f0:13.0-12 "int forks[n]";
  5->6: i = 0 at f0:28.7-16 "int i = 0";
--> State 1

Transition 2: State 1, proc 0:
  6->8: LOOP_TRUE_BRANCH at f0:28.18-23 "i < n";
--> State 2

...

Transition 12: State 12, proc 2:
  18->19: left = id at f0:16.2-15 "int left = id";
  19->20: right = ((id+1)%n) at f0:17.2-26 "int right = (id ... n";
--> State 13

Transition 13: State 13, proc 2:
  20->23: LOOP_TRUE_BRANCH at f0:19.9-10 "1";
--> State 14

Transition 14: State 14, proc 1:
  23->25: forks[left] = 1 at f0:20.29-44 "forks[left] = 1";
--> State 15

Transition 15: State 15, proc 2:
  23->25: forks[left] = 1 at f0:20.29-44 "forks[left] = 1";
--> State 16

...
Violation of Deadlock found in State 16:
A deadlock is possible:
  Path condition: true
  Enabling predicate: false
ProcessState 0: terminated
ProcessState 1: at location 25, f0:21.30-42 "forks[right]"
  Enabling predicate: false
ProcessState 2: at location 25, f0:21.30-42 "forks[right]"
  Enabling predicate: false

Trace ends after 15 transitions.
Violation(s) found.
...

```

Figure 4.3: Output from `civl replay diningBad.cvl`

```

#include<civlc.h>
#define TAG 0
#define NPROCS 2
#define NTHREADS 2

$gcomm gcomm = $gcomm_create($here, NPROCS);

void MPI_Process (int rank) {
    $comm comm = $comm_create($here, gcomm, rank);
    $proc threads[NTHREADS];

    void Thread(int tid) {
        int x = rank;
        $message in, out = $message_pack(rank, 1-rank, TAG, &x, sizeof(int));

        for (int j=0; j<2; j++) {
            if (rank == 1) {
                for (int i=0; i<2; i++) $comm_enqueue(comm, out);
                for (int i=0; i<2; i++) in = $comm_dequeue(comm, 1-rank, TAG);
            } else {
                for (int i=0; i<2; i++) in = $comm_dequeue(comm, 1-rank, TAG);
                for (int i=0; i<2; i++) $comm_enqueue(comm, out);
            }
        }
    }

    for (int i=0; i<NTHREADS; i++) threads[i] = $spawn Thread(i);
    for (int i=0; i<NTHREADS; i++) $wait(threads[i]);
    $comm_destroy(comm);
}

void main() {
    $proc procs[NPROCS];

    for (int i=0; i<NPROCS; i++) procs[i] = $spawn MPI_Process(i);
    for (int i=0; i<NPROCS; i++) $wait(procs[i]);
    $gcomm_destroy(gcomm);
}

```

Figure 4.4: `mpi-pthreads.cvl`: CIVL-C model of a (defective) multithreaded MPI program.

Part II

Language

Chapter 5

Overview of CIVL-C

5.1 Main Concepts

CIVL-C is an extension of a subset of the C11 dialect of C. It includes the most commonly-used elements of C, including most of the syntax, types, expressions, and statements. Missing are some of the more esoteric type qualifiers, bitwise operations (at least for now), and much of the standard library. Moreover, none of the C11 language elements dealing with concurrency are included, as CIVL-C has its own concurrency primitives.

The keywords in CIVL-C not already in C begin with the symbol \$. This makes them readily identifiable and also prevents any naming conflicts with identifiers in C programs. This means that most legal C programs will also be legal CIVL-C programs.

One of the most important features of CIVL-C not found in standard C is the ability to define functions in any scope. (Standard C allows function definitions only in the file scope.) This feature is also found in GNU C, the GNU extension of C.

Another central CIVL-C feature is the ability to *spawn* functions, i.e., run the function in a new *process* (thread).

Scopes and *processes* are the two central themes of CIVL-C. Each has a static and a dynamic aspect. The static scopes correspond to the lexical scopes in the program—typically, regions delimited by curly braces `{...}`. At runtime, these scopes are *instantiated* when control in a process reaches the beginning of the scope. Processes are created dynamically by *spawning* functions; hence the functions are the static representation of processes.

5.2 Example Illustrating Scopes and Processes

To understand the static and dynamic nature of scopes and processes, and the relations between them, we consider the (artificial) example code of Figure 5.1. The static scopes in the scope are numbered from 0 to 6.

The static scopes have a tree structure: one scope is a child of another if the first is immediately contained in the second. Scope 0, which is the file scope (or *root* scope) is the root of this tree. The static scope tree is depicted in Figure 5.2 (left). Each scope is identified by its integer ID. Additionally, if the scope happens to be the scope of a function definition, the name of the function is included in this identifier. A node in this tree also shows the variables and functions declared in the scope. For brevity, we omit the *proc* variables.

We now look at what happens when this program executes. Figure 5.2 (right) illustrates a


```

0
int x;
void f() { 1
  int y;
  void f1() { 2 ...}
  if (x>0) { 3
    int z;
    void f2() { 4 int w; ...}
    $proc p2=$spawn f1();
    $proc p3=$spawn f1();
    $proc p4=$spawn f2();
    f2();
  }
}
void g() { 5 ... }
void main() { 6
  $proc p1=$spawn f();
  g();
}

```

Figure 5.1: CIVL-C code skeleton to illustrate scope hierarchy

possible state of the program at one point in an execution. We now explain how this state is arrived at.

First, there is an implicit *root function* placed around the entire code. The body of the *main* function becomes the body of the root function, and the *main* function itself disappears. This minor transformation does not change the structure of the scope tree.

Execution begins by spawning a process p_0 to execute the root function. This causes scope 0 to be instantiated. An instance of a static scope is known as a *dynamic scope*, or *dyscopes* for short. The dynamic scopes are represented by the ovals with double borders on the right side of Figure 5.2. Each dyscope specifies a value for every variable declared in the corresponding static scope. In this case, the value 3 has been assigned to variable x .

The state of process p_0 is represented by a *call stack* (green). The entries on this stack are *activation frames*. Each frame contains two data: a reference to a dyscope (indicated by blue arrows) and a current location (or programmer counter value) in the static scope corresponding to that dyscope (not shown). The dyscope defines the environment in which the process evaluates expressions and executes statements. The currently executing function of a process, corresponding to the top frame in the call stack, can “see” only the variables in its dyscope and those of all the ancestors of its dyscope in the dyscope tree.

Returning to the example, p_0 enters scope 6, instantiating that scope, and then spawns procedure f . This creates process p_1 , with a new stack with a frame pointing to a dyscope corresponding to static scope 1. The new process proceed to run concurrently with p_0 . Meanwhile, p_0 calls procedure g , which pushes a new entry onto its call stack, and instantiates scope 5. Hence p_0 has two entries on its stack: the bottom one pointing to the instance of scope 6, the top one pointing to the instance of scope 5.

Meanwhile, assume $x > 0$, so that p_1 takes the *true* branch of the *if* statement, instantiating scope 3 under the instance of scope 1. It then spawns two copies of procedure $f1$, creating processes p_2 and p_3 and two instances of scope 2. Then p_1 spawns $f2$, creating process p_4 and an instance of scope 4. Note that the instance of scope 4 is a child of the instance of scope 3, since the (static)

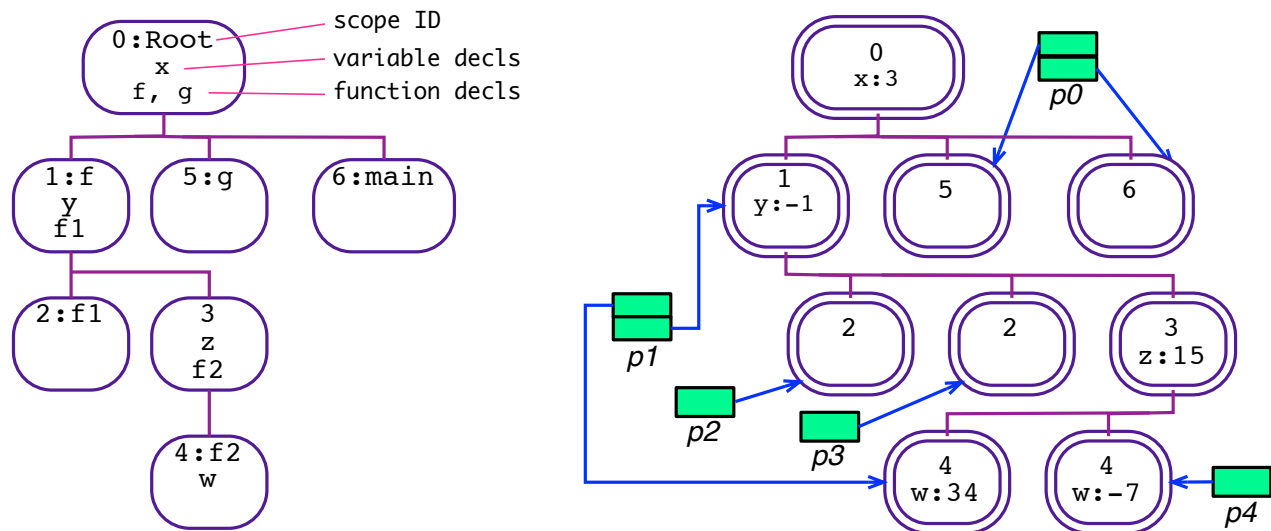


Figure 5.2: Static scope tree and a state for example program

scope 4 is a child of scope 3. Finally, p_1 calls `f2`, pushing a new entry on its stack and creating another instance of scope 4. The final state arrived at is the one shown.

There are few key points to understand:

- In any state, there is a mapping from the dyscope tree to the static scope tree which maps a dyscope to the static scope of which it is an instance. This mapping is a *tree homomorphism*, i.e., if dyscope u is a child of dyscope v , then the static scope corresponding to u is a child of the static scope corresponding to v .
- A static scope may have any number of instances, including 0.
- Dynamic scopes are created when control enters the corresponding static scope; they disappear from the state when they become unreachable. A dyscope v is “reachable” if some process has a frame pointing to a dyscope u and there is a path from u up to v that follows the parent edges in the dyscope tree.
- Processes are created when functions are spawned; they disappear from the state when their stack becomes empty (either because the process terminates normally or invokes the `exit` system function).

5.3 Structure of a CIVL-C program

A CIVL-C program is structured very much like a standard C program. In particular, a CIVL-C program may use the preprocessor directives specified in the C Standard, and with the same meaning. A source program is preprocessed, then parsed, resulting in a translation unit, just as with standard C. The main differences are the nesting of function definitions and the new primitives beginning with `$`, which are described in detail in the remainder of this part of the manual.

A CIVL-C program must begin with the line

```
#include <civlc.h>
```

which includes the main CIVL-C header file, which declares all the types and other CIVL primitives.

As usual, a translation unit consists of a sequence of variable declarations, function prototypes, and function definitions in file scope. In addition, *assume* statements may occur in the file scope. These are used to state assumptions on the input values to a program.

Chapter 6

Sequential Elements

In this chapter we describe the main sequential elements of the language. For the most part these are the same as in C. Primitives dealing with concurrency are introduced in Chapter 7.

6.1 Types

6.1.1 Standard types inherited from C

The boolean type is denoted `_Bool`, as in C. Its values are 0 and 1, which are also denoted by `$false` and `$true`, respectively.

There is one integer type, corresponding to the mathematical integers. Currently, all of the C integer types `int`, `long`, `unsigned int`, `short`, etc., are mapped to the CIVL integer type.

There is one real type, corresponding to the mathematical real numbers. Currently, all of the C real types `double`, `float`, etc., are mapped to the CIVL real type.

Array types, `struct` and `union` types, `char`, and pointer types (including pointers to functions) are all exactly as in C.

6.1.2 The bundle type: `$bundle`

CIVL-C includes a type named `$bundle`. A bundle is basically a sequence of data, wrapped into an atomic package. A bundle is created using a function that specifies a region of memory. One can create a bundle from an array of integers, and another bundle from an array of reals. Both bundles have the same type, `$bundle`. They can therefore be entered into an array of `$bundle`, for example. Hence bundles are useful for mixing objects of different (even statically unknown) types into a single data structure. Later, the contents of a bundle can be extracted with another function that specifies a region of memory into which to unpack the bundle; if that memory does not have the right type to receive the contents of the bundle, a runtime error is generated.

The relevant functions for creating and manipulating bundles are given in Figure 6.1.

6.1.3 The `$scope` type

An object of type `$scope` is a reference to a dynamic scope. It may be thought of as a “dynamic scope ID,” but it is not an integer and cannot be converted to an integer. Operations defined on scopes are discussed in Section 6.2.2.

```

/* Creates a bundle from the memory region specified by ptr and size,
 * copying the data into the new bundle */
$bundle $bundle_pack(void *ptr, int size);

/* Returns the size (number of bytes) of the bundle */
int $bundle_size($bundle b);

/* Copies the data out of the bundle into the region specified */
void $bundle_unpack($bundle bundle, void *ptr);

```

Figure 6.1: The *bundle* abstract data type

6.2 Expressions

6.2.1 Expressions inherited from C

The following C expressions are included in CIVL:

- *constant* expressions
- *identifier* expressions (x)
- *parenthetical* expressions ((e))
- numerical *addition* ($a+b$), *subtraction* ($a-b$), *multiplication* ($a*b$), *division* (a/b), *unary plus* ($+a$), *unary minus* ($-a$), *integer division* (a/b) and *modulus* ($a\%b$), all with their ideal mathematical interpretations
- array *index* expressions ($a[e]$) and struct or union *navigation* expressions ($x.f$, $p->f$)
- *address-of* ($\&e$), pointer *dereference* ($*p$), pointer *addition* ($p+i$) and *subtraction* ($p-q$) expressions
- relational expressions ($a==b$, $a!=b$, $a>=b$, $a<=b$, $a<b$, $a>b$)
- logical *not* ($!p$), *and* ($p\&\&q$), and *or* ($p||q$)
- *sizeof* a type ($sizeof(t)$) or expression ($sizeof(e)$)
- *assignment* expressions ($a=b$, $a+=b$, $a-=b$, $a*=b$, $a/=b$, $a\%=b$, $a++$, $a--$)
- function *calls* $f(e_1, \dots, e_n)$
- *conditional* expressions ($b ? e : f$).
- *cast* expressions ($(t)e$)

Bit-wise operations are not yet supported.

6.2.2 Scope expressions

As mentioned in Section 6.1.3, CIVL-C provides a type `$scope`. An object of this type is a reference to a dynamic scope. Several constants, expressions, and functions dealing with the `$scope` type are also provided.

The `$scope` type is like any other object type. It may be used as the element type of an array, a field in a structure or union, and so on. Expressions of type `$scope` may occur on the left or right-hand sides of assignments and as arguments in function calls just like any other expression. Two different variables of type `$scope` may be aliased, i.e., they may refer to the same dynamic scope.

A dynamic scope δ is *reachable* if there exists a path which starts from the dyscope referenced by some frame on the call stack of a process, follows the parent edges in the dyscope tree, and terminates in δ . If a dyscope is not reachable, it can never become reachable, and it cannot have any effect on the subsequent execution of the program.

Normally, a dynamic scope will eventually become unreachable. At some point after it becomes unreachable, it will be collected in a garbage-collection-like sweep, and any existing references to that scope will become *undefined*. An object of type `$scope` is also undefined before it is initialized. Any use of an undefined value is reported as an error by CIVL, so it is important to be sure that a scope variable is defined before using it.

6.2.2.1 Checking if a dyscope is defined: `$scope_defined`

The system function `$scope_defined` has signature

```
_Bool $scope_defined($scope s);
```

It returns *true* if the dynamic scope specified by `s` is defined, else it returns *false*.

6.2.2.2 The constant `$here`

A constant `$here` exists in every scope. This constant has type `$scope` and refers to the dynamic scope in which it is contained. For example,

```
{ // scope s
  int *p = (int*)$malloc($here, n*sizeof(int));
}
```

allocates an object consisting of n ints in the scope s .

6.2.2.3 The constant `$root`

There is a global constant `$root` of type `$scope` which refers to the root dynamic scope.

6.2.2.4 Scope relational operators

Let s_1 and s_2 be expressions of type `$scope`. The following are all CIVL-C expressions of boolean type:

- $s_1 == s_2$. This is *true* iff s_1 and s_2 refer to the same dynamic scope.
- $s_1 != s_2$. This is *true* iff s_1 and s_2 refer to different dynamic scopes.

- $s_1 \leq s_2$. This is *true* iff s_1 is equal to or a descendant of s_2 , i.e., s_1 is equal to or contained in s_2 .
- $s_1 < s_2$. This is *true* iff s_1 is a strict descendant of s_2 , i.e., s_1 is contained in s_2 and is not equal to s_2 .
- $s_1 > s_2$. This is equivalent to $s_2 < s_1$.
- $s_1 \geq s_2$. This is equivalent to $s_2 \leq s_1$.

If s_1 or s_2 is undefined in any of these expressions, an error will be reported.

6.2.2.5 Scope parent function `$scope_parent`

The system function

```
$scope $scope_parent($scope s);
```

returns the parent dynamic scope of the dynamic scope referenced by `s`. If `s` is the root dynamic scope, it returns the undefined value of type `$scope`.

6.2.2.6 Lowest Common Ancestor: `+`

The expression $s_1 + s_2$, where s_1 and s_2 are expressions of type `$scope`, evaluates to the lowest common ancestor of s_1 and s_2 in the dynamic scope tree. This is the smallest dynamic scope containing both s_1 and s_2 .

6.2.2.7 The `$scopeof` expression

Given any left-hand-side expression `expr`, the expression

```
$scopeof(expr)
```

evaluates to the dynamic scope containing the object specified by `expr`.

The following example illustrates the semantics of the `$scopeof` operator. All of the assertions hold:

```
{
  $scope s1 = $here;
  int x;
  double a[10];

  {
    $scope s2 = $here;
    int *p = &x;
    double *q = &a[4];

    assert($scopeof(x)==s1);
    assert($scopeof(p)==s2);
    assert($scopeof(*p)==s1);
    assert($scopeof(a)==s1);
```

```

    assert($scopeof(a[5])==s1);
    assert($scopeof(q)==s2);
    assert($scopeof(*q)==s1);
}
}

```

6.3 Statements

The usual C statements are supported:

- *no-op* (`;`)
- expression statements (`e;`)
- labeled statements, including `case` and `default` labels (`l: s`)
- *for* (`for (init; cond; inc) s`), *while* (`while (cond) s`) and *do* (`do s while (cond)`) loops
- compound statements (`{s1;s2; ...}`)
- `if` and `if ...else`
- `goto`
- `switch`
- `break`
- `continue`
- `return`

6.4 Guards and nondeterminism

6.4.1 Guarded commands: `$when`

A guarded command is encoded in CIVL-C using a `$when` statement:

```
$when (expr) stmt;
```

All statements have a guard, either implicit or explicit. For most statements, the guard is `$true`. The `$when` statement allows one to attach an explicit guard to a statement.

When `expr` is *true*, the statement is enabled, otherwise it is disabled. A disabled statement is *blocked*—it will not be scheduled for execution. When it is enabled, it may execute by moving control to the `stmt` and executing the first atomic action in the `stmt`.

If `stmt` itself has a non-trivial guard, the guard of the `$when` statement is effectively the conjunction of the `expr` and the guard of `stmt`.

The evaluation of `expr` and the first atomic action of `stmt` effectively occur as a single atomic action. There is no guarantee that execution of `stmt` will continue atomically if it contains more than one atomic action, i.e., other processes may be scheduled.

Examples:


```
$when (s>0) s--;
```

This will block until `s` is positive and then decrement `s`. The execution of `s--` is guaranteed to take place in an environment in which `s` is positive.

```
$when (s>0) {s--; t++}
```

The execution of `s--` must happen when `s>0`, but between `s--` and `t++`, other processes may execute.

```
$when (s>0) $when (t>0) x=y*t;
```

This blocks until both `x` and `t` are positive then executes the assignment in that state. It is equivalent to

```
$when (s>0 && t>0) x=y*t;
```

6.4.2 Nondeterministic selection statement: `$choose`

A `$choose` statement has the form

```
$choose {
  stmt1;
  stmt2;
  ...
  default: stmt
}
```

The `default` clause is optional.

The guards of the statements are evaluated and among those that are *true*, one is chosen nondeterministically and executed. If none are *true* and the `default` clause is present, it is chosen. The `default` clause will only be selected if all guards are *false*. If no `default` clause is present and all guards are *false*, the statement blocks. Hence the implicit guard of the `$choose` statement without a `default` clause is the disjunction of the guards of its sub-statements. The implicit guard of the `$choose` statement with a `default` clause is *true*.

Example: this shows how to encode a “low-level” CIVL guarded transition system:

```
11: $choose {
  $when (x>0) {x--; goto 12;}
  $when (x==0) {y=1; goto 13;}
  default: {z=1; goto 14;}
}
12: $choose {
  ...
}
13: $choose {
  ...
}
```

6.4.3 Nondeterministic choice of integer: \$choose_int

The system function \$choose_int has the following signature:

```
int $choose_int(int n);
```

This function takes as input a positive integer n and nondeterministically returns an integer in the range $[0, n - 1]$.

Chapter 7

Concurrency

7.1 Process creation and management

7.1.1 The process type: `$proc`

This is a primitive object type and functions like any other primitive C type (e.g., `int`). An object of this type refers to a process. It can be thought of as a process ID, but it is not an integer and cannot be cast to one. It is analogous to the `$scope` type for dynamic scopes.

Certain expressions take an argument of `$proc` type and some return something of `$proc` type. The operators `==` and `!=` may be used with two arguments of type `$proc` to determine whether the two arguments refer to the same process.

7.1.2 Checking if a process is defined: `$proc_defined`

An object of type `$proc` is initially undefined, so a use of that object would result in an error. One can check whether a `$proc` object is defined using the method `$proc_defined`:

```
_Bool $proc_defined($proc p);
```

7.1.3 Obtaining the null process reference: `$proc_null`

This function takes a pointer to an object of `$proc` type as a parameter:

```
void $proc_null($proc *p);
```

It updates the corresponding object with the default null value of the `$proc` type. The null value of `$proc` type is considered as defined, i.e., there will be no error to use an `$proc` object with null value.

7.1.4 The *self* process constant: `$self`

This is a constant of type `$proc`. It can be used wherever an argument of type `$proc` is called for. It refers to the process that is evaluating the expression containing `$self`.

7.1.5 Spawning a new process: `$spawn`

A *spawn* expression is an expression with side-effects. It spawns a new process and returns a reference to the new process, i.e., an object of type `$proc`. The syntax is the same as a procedure invocation with the keyword `$spawn` inserted in front:

```
$spawn f(expr1, ..., exprn)
```

Typically the returned value is assigned to a variable, e.g.,

```
$proc p = $spawn f(i);
```

If the invoked function `f` returns a value, that value is simply ignored.

7.1.6 Waiting for another process to terminate: `$wait`

The system function `$wait` has signature

```
void $wait($proc p);
```

When invoked, this function will not return until the process referenced by `p` has terminated. Note that `p` can be any expression of type `$proc`, not just a variable.

7.1.7 Terminating a process immediately: `$exit`

This function takes no arguments. It causes the calling process to terminate immediately, regardless of the state of its call stack:

```
void $exit(void);
```

7.2 Atomicity

7.2.1 Atom blocks: `$atom`

This defines a number of statements to be executed as a single atomic transition. An `$atom` block has the following form:

```
$atom {
    stmt1;
    stmt2;
    ...
}
```

The statements inside an `$atom` block are to be executed as one transition. It is required that the execution of the statements in an `$atom` block satisfy all of the following properties:

1. *deterministic*: at each step in the execution of the atom block, there must be at most one enabled statement;
2. *nonblocking*: at each step in the execution, there must be at least one enabled statement, hence, together with (1), there must be exactly one enabled statement;

3. *finite*: the execution of the atom block must terminate after a finite number of steps; and
4. *isolated*: there are no jumps from outside the atom block to inside the atom block, or from inside the atom block to outside of it.

Violations of the *deterministic*, *nonblocking*, or *isolated* properties will be reported either statically or dynamically. If the *finite* property is violated, the verification may just run forever.

Once the process enters an `$atom` block is said to be *executing atomly*. The process remains executing atomly until it reaches the terminating right brace of the block. Hence *executing atomly* is a dynamic, not static condition. For example, the block might contain a function call which takes the process to a point in code which is not statically contained in an atom block; that process is nevertheless still executing atomly and is subject to the rules above. The process only stops executing atomly when that function call returns and control finally reaches the right curly brace at the end of the atom block (assuming the block is not contained in another atom block).

Note: `$wait` statements are not allowed in `$atom` blocks. The rationale for this is that there is never a way to know for certain that another process has terminated (until `$wait` has returned) so there is never a way to be certain the `$wait` statement will not block. If one does occur in an `$atom` block, an error will be reported statically (if it can be detected statically) or dynamically (otherwise). Note that it is not always possible to detect this statically because the `$atom` block may contain a function call, and the function may contain the `$wait` statement.

7.2.2 Atomic blocks: `$atomic`

The statements in an *atomic* block will be executed without other processes interleaving, to the extent possible. It has the form:

```
$atomic {
    stmt1;
    stmt2;
    ...
}
```

It is essentially a weaker form of `$atom`. Unlike `$atom`, there are no restrictions on the statements that can go inside an `$atomic` block. A process executing an `$atomic` block will try to execute the statements without interleaving with other processes, unless it becomes blocked. Unlike an `$atom`, the statements in an atomic block do not necessarily execute as a single transition; they may be spread out over multiple transitions.

When no statement is enabled, the execution of the `$atomic` block will be interrupted. At this point, other processes are allowed to execute. Eventually, if the original process becomes enabled due to the actions of other processes, it may be scheduled again, in which case it regains atomicity and continues where it left off. For example, after executing the first loop, the process executing the following code will become blocked at the first `$wait` statement:

```
$atomic{
    for(int i = 0; i < 5; i++) p[i] = $spawn foo(i);
    for(int i = 0; i < 5; i++) $wait p[i];
}
```

Other processes will then execute. Eventually, if the process being waited on terminates, the original process becomes enabled and may be scheduled, in which case it regain atomicity, increments `i` and

proceeds to the next `$wait` statement. This is in fact a common idiom for spawning and waiting on a set of processes.

A process that enters an `$atomic` block is said to be *executing atomically*; it remains executing atomically until it reaches the closing curly brace.

Both `$atom` and `$atomic` blocks can be nested arbitrarily, but `$atom` overrides `$atomic`: a process that is executing atomly will continue executing atomly if it encounters an `$atomic` statement; but a process executing atomically that encounters an `$atom` will begin executing atomly.

The atomic semantics are defined more precisely as follows: there is a single global variable called the *atomic lock*. This variable can either be null (meaning the atomic lock is “free”), or it can hold the PID of a process; that process is said to “hold” the atomic lock. Moreover, each process contains a special integer variable, its *atomic counter*, which is initially 0. Every time a process enters an atomic block, it increments its atomic counter; every time it exits an atomic block, it decrements its counter. In order to increment its counter from 0 to 1, it must first wait for the atomic lock to become free, and then take the lock. When it decrements its counter from 1 to 0, it releases the atomic lock. When a process executing atomically becomes blocked, it releases the lock (without changing the value of its atomic counter).

7.3 Message-Passing

CIVL-C provides a number of additional primitives that can be used to model message-passing systems. This part of the language is built in two layers: the lower layer defines an abstract data type for representing messages; the higher layer defines an abstract data type of *communicators* for managing sets of messages being transferred among some set of processes.

7.3.1 Messages: `$message`

Messages are similar to bundles, but with some additional meta-data. The *data* component of the message is the “contents” of the message and is formed and extracted much like a bundle. The meta-data consists of an integer identifier for the *source* place of the message, an integer identifier for the message *destination* place, and an integer *tag* which can be used by a process to discriminate among messages for reception. This is very similar to MPI.

The functions for creating, and extracting information from, messages are given in Figure 7.1.

7.3.2 Communicators: `$gcomm` and `$comm`

CIVL-C defines a *global communicator* type `$gcomm` and a *local communicator* type `$comm`. The global communicator is an abstraction for a “communication universe” that stores buffered messages and perhaps other data. The local communicator wraps together a reference to a global communicator and an integer *place*. Most of the message-passing commands take a local communicator as an argument to specify the communication universe used for that operation and the place from which that operation will be executed. The communication universes are isolated from one another—a message sent on one can never be received using a different communicator, for example.

The global communicator is the shared object that must be declared in a scope containing all scopes in which communication in that universe will take place. It is created by specifying the number of *places* that will comprise the communicator. A place is an address to which messages may be sent or where they may be received. There is not necessarily a one-to-one correspondence between places and processes: many processes can use the same place.

```

/* creates a new message, copying data from the specified buffer */
$message_pack(int source, int dest, int tag, void *data, int size);

/* returns the message source */
int $message_source($message message);

/* returns the message tag */
int $message_tag($message message);

/* returns the message destination */
int $message_dest($message message);

/* returns the message size */
int $message_size($message message);

/* transfers message data to buf, throwing exception if message
 * size exceeds specified size */
void $message_unpack($message message, void *buf, int size);

```

Figure 7.1: The *message* abstract data type

Local communicators are created (typically in some child scope of the scope in which the global communicator is declared) by specifying the global communicator to which the local one will be associated and the place ID. The local communicator will be used in most of the message-passing functions; it may be thought of as an ordered pair consisting of a reference to the global communicator and the integer place ID. The place ID must be in $[0, \text{size} - 1]$, where **size** is the size of the global communicator. The place ID specifies the place in the global communication universe that will be occupied by the local communicator. The local communicator handle may be used by more than one process, but all of those processes will be viewed as occupying the same place. Only one call to `$comm_create` may occur for each `gcomm-place` pair.

Both types (`$gcomm` and `$comm`) are handle types. When declared with a call to the corresponding creation function, they create an object in the specified scope and return a handle to that object. The object can only be accessed through the specified system functions that take this handle as an argument.

The communicator interface is given in Figure 7.2.

Certain restrictions are enforced on some relations between the objects involved in a communication universe.

Fix a `$gcomm` object. This object corresponds to a single communication universe with, say, n places. At any time, there can be *at most one* `$comm` object associated to a given place. If a program attempts to create a `$comm` object with the same `$gcomm` and place as an earlier created `$comm` object, a runtime error will occur. In particular, there can be at most n `$comm` objects associated to the `$gcomm`.

The relation between processes and `$comm` objects is unconstrained. One process may use any number of `$comm` objects. (Of course, the process must have access to handles for those `$comm` objects.) Dually, a single `$comm` object may be used by any number of processes; this situation arises naturally when modeling a multi-threaded MPI program.

There is no special status given to the process which creates the `$comm` object of a given place.

```

/* Creates a new global communicator object and returns a handle to it.
 * The global communicator will have size communication places. The
 * global communicator defines a communication "universe" and encompasses
 * message buffers and all other components of the state associated to
 * message-passing. The new object will be allocated in the given scope. */
$gcomm $gcomm_create($scope s, int size);

void $gcomm_destroy($gcomm gcomm); // Destroys the gcomm

_Bool $gcomm_defined($gcomm gcomm); // Is the gcomm object defined?

/* Creates a new local communicator object and returns a handle to it.
 * The new communicator will be affiliated with the specified global
 * communicator. The new object will be allocated in the given scope. */
$comm $comm_create($scope s, $gcomm gcomm, int place);

void $comm_destroy($comm comm); // Destroys the comm

_Bool $comm_defined($comm comm); // Is the comm object defined?

/* Returns the size (number of places) in the global communicator associated
 * to the given comm. */
int $comm_size($comm comm);

/* Returns the place of the local communicator. This is the same as the
 * place argument used to create the local communicator. */
int $comm_place($comm comm);

/* Adds the message to the appropriate message queue in the communication
 * universe specified by the comm. The source of the message must equal
 * the place of the comm. */
void $comm_enqueue($comm comm, $message message);

/* Returns true iff a matching message exists in the communication universe
 * specified by the comm. A message matches the arguments if the destination
 * of the message is the place of the comm, and the sources and tags match. */
_Bool $comm_probe($comm comm, int source, int tag);

/* Finds the first matching message and returns it without modifying
 * the communication universe. If no matching message exists, returns a message
 * with source, dest, and tag all negative. */
$message $comm_seek($comm comm, int source, int tag);

/* Finds the first matching message, removes it from the communicator,
 * and returns the message */
$message $comm_dequeue($comm comm, int source, int tag);

```

Figure 7.2: The *communicator* interface specifies handle types `$gcomm` and `$comm` and the functions above


```
$gcomm gcomm = $gcomm_create($here, nprocs);
void Process(int rank) {
    $comm comm = $comm_create($here, gcomm, rank);

    void Thread(int tid) {
        ...$comm_enqueue(comm, msg)...
        ...$comm_dequeue(comm, source, tag)...
    }

    for (int i=0; i<nthreads; i++) $spawn Thread(i);
    ...
    $comm_destroy(comm);
}
for (int i=0; i<nprocs; i++) $spawn Process(i);
...
$gcomm_destroy(gcomm);
```

Figure 7.3: Code skeleton for model of multithreaded MPI program showing placement of global and local communicator objects

Any process which can access a handle for that `$comm` object can use it to send or receive messages, regardless of whether that process was the one that created the `$comm` object. However, users should be aware that verification is likely to be most efficient when variables are declared as locally as possible, so it is best to declare the `$comm` object in the innermost scope possible. Figure 7.3 illustrates an effective way to do this in the context of modeling a multithreaded MPI program. In the code skeleton, each thread can access the local communicator object of its process, but not that of any other process.

Chapter 8

Specification

8.1 Overview

Specification is the means by which one expresses what a program is supposed to do, i.e., what it means for it to be correct.

There are several specification mechanisms in CIVL-C. First, there are the default properties: these are generic properties which are checked by default in any program, and require no additional specification effort. These properties include absence of deadlocks, division by 0, illegal pointer dereferences, and out of bounds array indexes.

Many more program-specific properties can be specified using assertions. CIVL-C has a rich assertion language which extends the language of boolean-valued C expressions. Assumptions are a specification dual to assertions in that they restrict the set of executions on which the assertions are checked.

Functional equivalence is a power specification mechanism. In this approach, two programs are provided, one playing the role of the specification, the other the role of the implementation. The implementation is correct if, for all inputs x , it produces the same output as that produced by the specification on input x . In other words, the two programs define the same function; this is sometimes known as *input-output equivalence*. In order to take this approach, one must first have a way to specify what the inputs and outputs of a programs are; CIVL-C provides special keywords for this.

Procedure contracts are another powerful specification mechanisms. These typically involve specifying preconditions and postconditions for a function. The function is correct if, whenever it is called in a state satisfying the precondition, when it returns the state will satisfy the postcondition. A program is correct if all its functions satisfy their contract.

8.2 Input-output signature

8.2.1 Input type qualifier: `$input`

The declaration of a variable in the root scope may include the type qualifier `$input`, e.g.,

```
$input int n;
```

This declares the variable to be an input variable, i.e., one which is considered to be an input to the program. Such a variable is initialized with an arbitrary (unconstrained) value of its type. When

using symbolic execution to verify a program, such a variable will be assigned a unique symbolic constant of its type.

In contrast, variables in the root scope which are not input variables will instead be initialized with the “undefined” value. If an undefined value is used in some way (such as in an argument to an operator), an error occurs.

In addition, input variables may only be read, never written to.

Alternatively, it is also possible to specify a particular concrete initial value for an input variable. This is done using a command line argument when verifying or running the program.

Input (and output) variables also play a key role when determining whether two programs are functionally equivalent. Two programs are considered functionally equivalent if, whenever they are given the same inputs (i.e., corresponding `$input` variables are initialized with the same values) they will produce the same outputs (i.e., corresponding `$output` variables will end up with the same values at termination).

8.2.2 Output type qualifier: `$output`

A variable in the root scope may be declared with this type qualifier to declare it to be an output variable. Output variables are “dual” to input variables. They may only be written to, never read. They are used primarily in functional equivalence checking.

8.3 Assertions and assumptions

8.3.1 Assertions: `$assert`

The system function `$assert` takes an argument of boolean type:

```
void $assert (_Bool expr);
```

During verification, the assertion is checked. If it cannot be proved that it must hold, a violation is reported.

Note that CIVL-C boolean expressions have a richer syntax than C expressions, and may include universal or existential quantifiers (see below), and the boolean values `$true` and `$false`.

The assertion function may take additional optional arguments used to print a specific message if the assertion is violated. These additional arguments are similar in form to those used in C’s `printf` statement: a format string, followed by some number of arguments which are evaluated and substituted for successive codes in the format string. For example,

```
$assert(x<=B, "x-coordinate %f exceeds bound %f", x, B);
```

The C function `assert`, included in the standard library `assert.h`, is identical to `$assert`. Programmers are free to use either one.

8.3.2 Assume statements: `$assume`

As *assume statement* has the form

```
$assume expr;
```

During verification, the assumed expression is assumed to hold. If this leads to a contradiction on some execution, that execution is simply ignored. It never reports a violation, it only restricts the set of possible executions that will be explored by the verification algorithm.

Like an assertion statement, an assume statement can be used anywhere a statement is expected. In addition, an assume statement can be used in file scope to place restrictions on the global variables of the programs. For example,

```
$input int B;
$input int N;
$assume 0<=N && N<=B;
```

declares `N` and `B` to be integer inputs and restricts consideration to inputs satisfying $0 \leq N \leq B$.

8.4 Formulas

A formula is a boolean expression that can be used in an assert statement, assume statement, procedure contract (below), or invariant. Any ordinary C boolean expression is a formula. CIVL-C provides some additional kinds of formulas, described below.

8.4.1 Implication: `=>`

The binary operation `=>` represents logical implication. The expression `p=>q` is equivalent to `(!p)||q`.

8.4.2 Universal quantifier: `$forall`

The universally quantified formula has the form

```
$forall { type identifier | restriction } expr
```

where `type` is a type name (e.g., `int` or `double`), `identifier` is the name of the bound variable, `restriction` is a boolean expression which expresses some restriction on the values that the bound variable can take, and `expr` is a formula. The universally quantified formula holds iff for all values assignable to the bound variable for which the restriction holds, the formula `expr` holds.

A variation on the construct above can be used in the special case where the bound variable is to range over a finite interval of integers. In this case the quantified formula may be written:

```
$forall { type identifier=lower .. upper } expr
```

where `lower` and `upper` are integer expressions.

8.4.3 Existential quantifier: `$exists`

The syntax for existentially quantified expressions is exactly the same as for universally quantified expressions, with `$exists` in place of `$forall`.

8.5 Contracts

8.5.1 Procedure contracts: `$requires` and `$ensures`

The `$requires` and `$ensures` primitives are used to encode procedure contracts. There are optional elements that may occur in a procedure declaration or definition, as follows. For a function prototype:

```
T f(...)
  $requires expr;
  $ensures expr;
;
```

For a function definition:

```
T f(...)
  $requires expr;
  $ensures expr;
{
  ...
}
```

The value `$result` may be used in post-conditions to refer to the result returned by a procedure.

Status: parsed, but nothing is currently done with this information.

8.5.2 Loop invariants: `$invariant`

This indicates a loop invariant. Each C loop construct has an optional invariant clause as follows:

```
while (expr) $invariant (expr) stmt
for (e1; e2; e3) $invariant (expr) stmt
do stmt while (expr) $invariant (expr) ;
```

The invariant encodes the claim that if `expr` holds upon entering the loop and the loop condition holds, then it will hold after completion of execution of the loop body. The invariant is used by certain verification techniques.

Status: parsed, but nothing is currently done with this information.

8.6 Concurrency specification

8.6.1 Remote expressions: `e@x`

These have the form `expr@x` and refer to a variable in another process, e.g., `procs[i]@x`. This special kind of expression is used in collective expressions, which are used to formulate collective assertions and invariants.

The expression `expr` must have `$proc` type. The variable `x` must be a statically visible variable in the context in which it occurs. When this expression is evaluated, the evaluation context will be shifted to the process referred to by `expr`.

Status: not implemented.

8.6.2 Collective expressions: `$collective`

. These have the form

```
$collective(proc_expr, int_expr) expr
```

This is a collective expression over a set of processes. The expression `proc_expr` yields a pointer to the first element of an array of `$proc`. The expression `int_expr` gives the length of that array, i.e., the number of processes. Expression `expr` is a boolean-valued expression; it may use remote expressions to refer to variables in the processes specified in the array. Example:

```
$proc procs[N];  
...  
$assert $collective(procs, N) i==procs[(pid+1)%N]@i ;
```

Status: not implemented.

Chapter 9

Pointers and Heaps

CIVL-C supports pointers, using the same operators with the same meanings as C (&, *, pointer arithmetic). There is also a heap in every scope, and system functions to allocate and deallocate objects in the specified scope.

9.1 Memory functions: memcpy

The function `memcpy` is defined in the standard C library `string.h` and works exactly the same in CIVL-C: it copies data from the region pointed to by `q` to that pointed to by `p`. The signature is

```
void memcpy(void *p, void *q, size_t size);
```

9.2 Heaps, `$malloc` and `$free`

As mentioned above, each dynamic scope has an implicit heap on which objects can be allocated and deallocated dynamically. To allocate an object, one first needs a reference to the dynamic scope to be used. The system function `$malloc` is like C's `malloc`, but takes this extra scope argument:

```
void * $malloc($scope scope, int size);
```

The standard C function

```
void * malloc(int size);
```

declared in `stdlib.h`, is equivalent to `$malloc($root, size)`.

The system function `$free` is used to deallocate a heap object; it is just like C's `free`:

```
void $free(void *p);
```

An error is generated if the pointer is not one that was returned by `$malloc`, or if it was already freed. The standard C function `free`, declared in `stdlib.h` is identical to `$free`. The two functions are interchangeable.

Chapter 10

Libraries

Each of the following libraries is at least partially implemented and can be included in a CIVL-C program:

- `assert`
 - `void assert(_Bool expr);`
- `math`
 - `double sqrt(double x);`
 - `double ceil(double x);`
 - `double exp(double x);`
- `stdlib`
 - `size_t`
 - `void * malloc(size_t size);`
 - `void free(void * ptr);`
- `stdbool`
 - `true`
 - `false`
- `stddef`
 - `size_t`
 - `NULL`
- `stdio`
 - `int printf(const char * restrict format, ...);`
- `string`
 - `size_t`
 - `NULL`
 - `void memcpy(void * restrict dst, const void * restrict src, size_t n);`

Part III

Semantics

Chapter 11

CIVL Model Syntax

11.1 Notation and terminology

Let $\mathbb{B} = \{true, false\}$ (the set of boolean values). Let $\mathbb{N} = \{0, 1, 2, \dots\}$ (the set of natural numbers).

Given a node u in a tree, we let $\text{ancestors}(u)$ denote the set of all ancestors of u , including u . We let $\text{descendants}(u)$ denote the set of all descendants of u , including u .

For any set S , let S^* denote the set of all finite sequences of elements of S . The length of a sequence $\xi \in S^*$ is denoted $\text{length}(\xi)$.

11.2 Definition of Context

Definition 11.2.0.1. A *CIVL type system* is a tuple comprising the following components:

1. a set Type (the set of *types*),
2. a type $\text{bool} \in \text{Type}$ (the *boolean type*),
3. a type $\text{proc} \in \text{Type}$ (the *process-reference type*),
4. a set Var (the set of all *typed variables*),
5. a function $\text{vtype}: \text{Var} \rightarrow \text{Type}$ (which gives the type of each variable),
6. a set Val (the set of all *values*),
7. a function which assigns to each $t \in \text{Type}$ a subset $\text{Val}_t \subseteq \text{Val}$ (the set of values of type t) and which satisfies $\text{Val}_{\text{bool}} = \mathbb{B}$ and $\text{Val}_{\text{proc}} = \mathbb{N}$,
8. a function which assigns to each $t \in \text{Type}$ a value $\text{default}_t \in \text{Val}_t$.

The default value will be used to give an initial value to any variable. It could represent something like “an undefined value of type t ” or a reasonable initial value (0 for integers, etc.), depending on the language one is modeling.

Definition 11.2.0.2. Given a CIVL type system, a *valuation* in that system is a function $\eta: \text{Var} \rightarrow \text{Val}$ with the property that for any $v \in \text{Var}$, $\eta(v) \in \text{Val}_{\text{vtype}(v)}$.

Given a CIVL type system, we let Eval denote the set of all valuations in that system.

Definition 11.2.0.3. Given a CIVL type system, A *CIVL expression system* for that type system is a tuple comprising the following components:

1. a set Expr (the set of all *typed expressions* over Var),
2. a function $\text{etype}: \text{Expr} \rightarrow \text{Type}$ (giving the type of each expression),
3. a function $\text{eval}: \text{Expr} \times \text{Eval} \rightarrow \text{Val}$ (the *evaluation function*), satisfying
 - for any $e \in \text{Expr}$ and $\eta \in \text{Eval}$, $\text{eval}(e, \eta) \in \text{Val}_{\text{etype}(e)}$,
4. a function which associates to any $V \subseteq \text{Var}$, a subset $\text{Expr}(V) \subseteq \text{Expr}$ (the set of *expressions which involve only variables in V*) satisfying the following:
 - for any $V \subseteq \text{Var}$ and $\eta, \eta' \in \text{Eval}$, if $\eta(v) = \eta'(v)$ for all $v \in V$, then for any $e \in \text{Expr}(V)$, $\text{eval}(e, \eta) = \text{eval}(e, \eta')$

Definition 11.2.0.4. A *CIVL context* is a CIVL type system together with a CIVL expression system for that type system.

11.3 Lexical scopes

Definition 11.3.0.5. Given a CIVL context $\mathcal{C}$, a *lexical scope system* over $\mathcal{C}$ is a tuple

$$(\Sigma, \text{root}, \text{sparent}, \text{vars})$$

consisting of

1. a set Σ (the set of *static scopes*),
2. a scope $\text{root} \in \Sigma$ (the *root scope*),
3. a function $\text{sparent}: \Sigma \setminus \{\text{root}\} \rightarrow \Sigma$ such that

$$\{(\text{sparent}(\sigma), \sigma) \mid \sigma \in \Sigma \setminus \{\text{root}\}\}$$

gives Σ the structure of a rooted tree with root root ,

4. a function $\text{vars}: \Sigma \rightarrow 2^{\text{Var}}$ (specifying the variables *declared* in each scope) satisfying
 - $\sigma \neq \tau \Rightarrow \text{vars}(\sigma) \cap \text{vars}(\tau) = \emptyset$.

Definition 11.3.0.6. Given a CIVL context and scope $\sigma \in \Sigma$, the set of *visible variables* in σ is $\bigcup_{\sigma' \in \text{ancestors}(\sigma)} \text{vars}(\sigma')$.

One way this notion will be used: expressions used in a scope σ can only involve variables visible in σ .

Symbol	Section	Meaning
$\mathbb{B}$	§11.1	$\{true, false\}$
$\mathbb{N}$	§11.1	$\{0, 1, 2, \dots\}$
ancestors	§11.1	set of ancestors of node in a tree (inclusive)
descendants	§11.1	set of descendants of node in a tree (inclusive)
length	§11.1	length of a sequence
Var	§11.2	the set of all variables
bool	§11.2	the boolean type
proc	§11.2	the process reference type
Val	§11.2	the set of all values
Val_t	§11.2	values of type t
default _{t}	§11.2	default value of type t
vtype	§11.2	function $Var \rightarrow Type$ giving type of each variable
Eval	§11.2	set of all valuations on Var
Eval(V)	§12.1	set of all valuations on variables in $V \subseteq Var$
Expr	§11.2	set of typed expressions over Var
etype	§11.2	$Expr \rightarrow Type$, gives type of each expression
eval	§11.2	$Expr \times Eval \rightarrow Val$, the evaluation function
$\mathcal{C}$	§11.2	a CIVL context
Σ	§11.3	set of all static scopes
root	§11.3	the root scope (member of Σ)
sparent	§11.3	$\Sigma \setminus \{\text{root}\} \rightarrow \Sigma$, parent function in static scope tree
vars	§11.3	$\Sigma \rightarrow 2^{Var}$, specifies variables declared in scope
Λ	§11.3	a lexical scope system
void	§11.4	type used for function that does not return a value
Type'	§11.4	$Type \cup \{\text{void}\}$
$\mathcal{F}$	§11.4	set of function symbols
fscope	§11.4	$\mathcal{F} \rightarrow \Sigma \setminus \{\text{root}\}$, gives function scope of each function
returnType	§11.4	$\mathcal{F} \rightarrow Type'$, gives return type of each function
params	§11.4	$\mathcal{F} \rightarrow Var^*$, formal parameter sequence for $f \in \mathcal{F}$
f_0	§11.4	the root function (member of $\mathcal{F}$)
func	§11.4	$\Sigma \rightarrow \mathcal{F}$, function to which scope belongs
Loc_f	§11.7	set of locations for $f \in \mathcal{F}$
lscope _{f}	§11.7	$Loc_f \rightarrow \Sigma$, gives scope of each location for $f \in \mathcal{F}$
start _{f}	§11.7	start location for $f \in \mathcal{F}$ (member of Loc_f)
T_f	§11.7	set of guarded transitions for $f \in \mathcal{F}$

Figure 11.1: Table of Notation Used to Define CIVL Model Syntax

11.4 Functions

Fix a CIVL context $\mathcal{C}$ and lexical scope system

$$\Lambda = (\Sigma, \text{root}, \text{sparent}, \text{vars})$$

over $\mathcal{C}$.

We introduce a new type symbol **void**, as in $\mathcal{C}$, to use as the return type for a function that does not return a value. Let $\text{Type}' = \text{Type} \cup \{\text{void}\}$.

Definition 11.4.0.7. A *function prototype* for Λ is a tuple (σ, t, ξ) consisting of

1. a scope $\sigma \in \Sigma \setminus \{\text{root}\}$ (the *function scope*),
2. a type $t \in \text{Type}'$ (the *return type*),
3. a finite sequence $\xi = v_1 v_2 \cdots v_n \in \text{vars}(\sigma)^*$ consisting of variables declared in the function scope (the *formal parameters*).

Definition 11.4.0.8. A *CIVL function prototype system* consists of

1. a set $\mathcal{F}$ (the *function symbols*),
2. a function which assigns to each $f \in \mathcal{F}$ a function prototype, denoted

$$(\text{fscope}(f), \text{returnType}(f), \text{params}(f)),$$

and satisfying

- for any $\sigma \in \Sigma$, there is at most one $f \in \mathcal{F}$ such that $\sigma = \text{fscope}(f)$, and
- 3. a *root function* f_0 with $\text{fscope}(f_0) = \text{root}$ and which is the only function with root scope.

Definition 11.4.0.9. Given a CIVL function prototype system, and function symbol $f \in \mathcal{F} \setminus \{f_0\}$, the *declaration scope* of f is the scope $\sigma = \text{sparent}(\text{fscope}(f))$. We also write f is *declared in* σ .

Note the root function f_0 has no declaration scope.

Just as every scope has a set of visible variables, there is also a set of visible functions:

Definition 11.4.0.10. The functions *visible at scope* σ are those declared in σ or an ancestor of σ .

We will see that the variables and functions visible at σ are the only variables and functions that can be referred to by statements and expressions used within σ .

Note that only certain scopes are function scopes. There can be additional scopes (intuitively corresponding to block scopes in a source program). Every scope, however, must “belong to” exactly one function. The precise definition is as follows:

Definition 11.4.0.11. Given a CIVL function prototype system, define

$$\text{func}: \Sigma \rightarrow \mathcal{F}$$

by

$$\text{func}(\sigma) = \begin{cases} f & \text{if } \sigma = \text{fscope}(f) \text{ for some } f \in \mathcal{F} \\ \text{func}(\text{sparent}(\sigma)) & \text{otherwise.} \end{cases}$$

We say σ *belongs to* f when $\text{func}(\sigma) = f$.

Note that the recursion in Definition 11.4.0.11 must stop as the root scope belongs to the root function and the scopes form a tree.

11.5 Statements

Fix a CIVL function prototype system. A *CIVL statement* is defined to be a tuple of one of the forms described below. In each case, we give any restrictions on the components of the tuple and a brief intuition on the statement's semantics. The precise semantics will be described in §12.

1. $\langle \text{parassign}, V_1, V_2, \psi \rangle$
 - $V_1, V_2 \subseteq \text{Var}$
 - $\psi: V_2 \rightarrow \text{Expr}(V_1)$ satisfying $\text{etype}(\psi(v)) = \text{vtype}(v)$ for all $v \in V_2$
 - *meaning*: parallel assignment, i.e., the assignment of new values to any or all of the variables in V_2 . For each variable in V_2 an expression is given which will be evaluated in the old state to compute the new value for that variable. V_1 contains all the variables that may be used in those expressions. Hence V_1 is the “read set” and V_2 is the “write set” for the parallel assignment.
2. $\langle \text{assign}, v, e \rangle$
 - $v \in \text{Var}, e \in \text{Expr}, \text{etype}(e) = \text{vtype}(v)$
 - *meaning*: simple assignment: evaluate an expression e and assign result to variable v . It is a special case of `parassign` but is provided for convenience.
3. $\langle \text{call}, y, f, e_1, \dots, e_n \rangle$
 - $y \in \text{Var}, f \in \mathcal{F}, e_1, \dots, e_n \in \text{Expr}$
 - $n = \text{numParams}(f)$
 - $\text{etype}(e_i) = \text{vtype}(v_i)$, where $\text{params}(f) = v_1 \cdots v_n$
 - $\text{returnType}(f) = \text{vtype}(y)$
 - *meaning*: evaluate expressions $e_1, \dots, e_n$; push frame on call stack and move control to guarded transition system (see §11.7) for function f ; when f returns, pop stack and store returned result in y
4. $\langle \text{call}, f, e_1, \dots, e_n \rangle$
 - $f \in \mathcal{F}, e_1, \dots, e_n \in \text{Expr}$
 - $n = \text{numParams}(f)$
 - $\text{etype}(e_i) = \text{vtype}(v_i)$, where $\text{params}(f) = v_1 \cdots v_n$
 - *meaning*: like above, but return type may be `void` or returned value could just be ignored
5. $\langle \text{fork}, p, f, e_1, \dots, e_n \rangle$
 - $p \in \text{Var}, f \in \mathcal{F}, e_1, \dots, e_n \in \text{Expr}$
 - $n = \text{numParams}(f)$
 - $\text{etype}(e_i) = \text{vtype}(v_i)$, where $\text{params}(f) = v_1 \cdots v_n$
 - $\text{returnType}(f) = \text{void}$
 - $\text{vtype}(p) = \text{proc}$

- *meaning*: evaluate expressions $e_1, \dots, e_n$; create new process and invoke function f in it; return, immediately, a reference to the new process and store that reference in p

6. $\langle \text{join}, e \rangle$

- $e \in \text{Expr}$
- $\text{etype}(e) = \text{proc}$
- *meaning*: evaluate e and wait for the referenced process to terminate

7. $\langle \text{return}, e \rangle$

- $e \in \text{Expr}$
- *meaning*: evaluate e , pop the call stack and return control, along with the value, to caller

8. $\langle \text{return} \rangle$

- *meaning*: pop the call stack and return control to caller; only to be used in functions returning `void`

9. $\langle \text{write}, e \rangle$

- $e \in \text{Expr}$
- *meaning*: evaluate e and send result to output

10. $\langle \text{noop} \rangle$

- *meaning*: does nothing

11. $\langle \text{assert}, e \rangle$

- $e \in \text{Expr}$, $\text{vtype}(e) = \text{bool}$
- *meaning*: evaluate e ; if result is false, stop execution and report error

12. $\langle \text{assume}, e \rangle$

- $e \in \text{Expr}$, $\text{vtype}(e) = \text{bool}$
- *meaning*: assume e holds (i.e., if e does not hold, the execution sequence is not a real execution)

11.6 Remarks

The system described is sufficiently general to model pointers. There can be (one or more) pointer types and corresponding values. The parallel assignment statement can be used to model statements like C's `*p=e;`. In the worst case (if no information is known about where `p` could point), one can let $V_2 = \text{Var}$. Similarly, expressions that involve `*p` as a right-hand side subexpression can always take $V_1 = \text{Var}$.

Heaps can also be modeled. A heap type may be defined and a variable of that type declared. Expressions to modify and read from the heap can be defined, as can pointers into the heap.

11.7 Transition system representation of functions

Definition 11.7.0.12. Given a CIVL function prototype system and $f \in \mathcal{F}$, a *guarded transition system* for f is a tuple $(\text{Loc}, \text{lscope}, \text{start}, T)$, where

- Loc is a set (the set of *locations*),
- $\text{lscope}: \text{Loc} \rightarrow \Sigma$ is a function which associates to each $l \in \text{Loc}$ a scope $\text{lscope}(l) \in \Sigma$ belonging to f ,
- $\text{start} \in \text{Loc}$ (the *start location*),
- T is a set of *guarded transitions*, each of which has the form $\langle l, g, s, l' \rangle$, where
 - $l, l' \in \text{Loc}$ (the *source* and *target* locations)
 - $g \in \text{Expr}(V)$, where V is the set of variables visible at $\text{lscope}(l)$, and $\text{etype}(g) = \text{bool}$ (g is called the *guard*),
 - s is a statment all of whose constituent variables, expressions, and function symbols are visible at $\text{lscope}(l)$.

Furthermore, if the guarded transition system contains a statement of the form $\langle \text{return} \rangle$ then $\text{returnType}(f) = \text{void}$. If it contains a statement of the form $\langle \text{return}, e \rangle$ then

$$\text{etype}(e) = \text{returnType}(f).$$

Definition 11.7.0.13. Given a CIVL prototype system, a *CIVL model* M for that system assigns, to each $f \in \mathcal{F}$, a guarded transition system

$$(\text{Loc}_f, \text{lscope}_f, \text{start}_f, T_f)$$

for f . Moreover, if $f \neq f'$ then $\text{Loc}_f \cap \text{Loc}_{f'} = \emptyset$.

Definition 11.7.0.14. Given a CIVL model M , let $\text{Loc} = \bigcup_{f \in \mathcal{F}} \text{Loc}_f$.

Chapter 12

CIVL Model Semantics

12.1 State

Fix a CIVL model M . Recall that a valuation is a type-respecting function from Var to Val . Given a subset $V \subseteq \text{Var}$ of variables, we define a *valuation on V* to be a type-respecting function from V to Val . Let $\text{Eval}(V)$ denote the set of all valuations on V . Note that $\text{Eval}(\text{Var}) = \text{Eval}$.

Definition 12.1.0.15. A *state* of a CIVL model M is a tuple

$$s = (\Delta, \text{droot}, \text{dparent}, \text{static}, \text{deval}, P, \text{stack}),$$

where

1. Δ is a finite set (the set of *dynamic scopes* in s),
2. $\text{droot} \in \Delta$ (the *root dynamic scope*),
3. $\text{dparent}: \Delta \setminus \{\text{droot}\} \rightarrow \Delta$ is a function such that the set

$$\{(\text{dparent}(\delta), \delta) \mid \delta \in \Delta \setminus \{\text{droot}\}\}$$

gives Δ the structure of a rooted tree with root droot ,

4. $\text{static}: \Delta \rightarrow \Sigma$,
5. $\text{static}(\text{droot}) = \text{root}$ and droot is the only $\delta \in \Delta$ satisfying $\text{static}(\delta) = \text{root}$,
6. $\text{static}(\text{dparent}(\delta)) = \text{sparent}(\text{static}(\delta))$ for any $\delta \in \Delta$,
7. deval is a function that assigns to each $\delta \in \Delta$ a valuation $\text{deval}(\delta) \in \text{Eval}(\text{vars}(\text{static}(\delta)))$,
8. P (the set of *process IDs* in s) is a finite subset of Val_{proc} , and
9. $\text{stack}: P \rightarrow \text{Frame}^*$, where

$$\text{Frame} = \{(\delta, l) \in \Delta \times \text{Loc} \mid \text{lscope}(l) = \text{static}(\delta)\}.$$

Let State denote the set of all states of M .

Remarks:

1. We will also refer to dynamic scopes as *dyscopes*.
2. The elements of Δ contain no intrinsic information. Instead, all of the information concerning dyscopes is encoded in the functions that take elements of Δ as arguments. Hence we might just as well call the elements of Δ “dynamic scope IDs” (just as we call the elements of P “process IDs”). One could use natural numbers for the dyscopes, just as one does for processes.
3. The reason for using some form of ID for dyscopes and processes, rather than just incorporating the data in the appropriate part of the state, is that both kinds of objects may be shared. There can be several components of the state that refer to the same dyscope d : e.g., d could have several children, each of which has a parent reference to d , as well as a reference from a frame. A process can be referred to by any number of variables of type `proc`.
4. If $\sigma = \text{static}(\delta)$, we say that δ is an instance of σ or σ is the static scope associated to δ . In general, a static scope can have any number (including 0) of dynamic instances. The exception is the root scope `root`, which must have exactly one instance (`droot`).
5. A valuation `deval`(δ) assigns a value to each variable in the static scope associated to δ . The function `deval` thereby encodes the value of all variables “in scope” in state s .
6. The sequence `stack`(p) encodes the state of the call stack of process p . The elements of the sequence are called *activation frames*. The first frame in the sequence, i.e., the frame in position 0, is the bottom of the stack; the last frame is the top of the stack.
7. Each frame refers to a dyscope δ and a location in the static scope associated to δ .

Definition 12.1.0.16. A dyscope $\delta \in \Delta$ is a *function node* if `static`(δ) is the function scope of some function.

Definition 12.1.0.17. Given any $\delta \in \Delta$, `fnode`(δ) $\in \Delta$ is defined as follows: if δ is a function node, then `fnode`(δ) = δ , else `fnode`(δ) = `fnode`(`dparent`(δ)). We call `fnode`(δ) the *function node associated to* δ .

The relation $\{(\delta, \delta') \mid \text{fnode}(\delta) = \text{fnode}(\delta')\}$ is an equivalence relation $\sim$ on Δ . Let $\bar{\Delta} = \Delta / \sim$ and write $[\delta]$ for the equivalence class containing δ .

The set of activation frames in a state s may be identified with the set

$$AF(s) = \bigcup_{p \in P} \{p\} \times \{0, \dots, \text{length}(\text{stack}(p)) - 1\}$$

Namely, (p, i) corresponds to the i^{th} entry in the call stack `stack`(p) (where the elements of the stacks are indexed from 0).

Define $\Psi: AF(s) \rightarrow \bar{\Delta}$ as follows: given (p, i) , let (δ, l) be the corresponding frame; set $\Psi(p, i) = [\delta]$.

12.2 Jump protocol

When control moves from one location to another within a function’s transition system, dyscopes may be added, because you can jump out of scope nests and into new scope nests. The motivating

```

1 procedure jump( $s$ : State,  $p$ : Valproc,  $l'$ : Loc): State is
2   let  $(\Delta, \text{droot}, \text{dparent}, \text{static}, \text{deval}, P, \text{stack}) = s$ ;
3   let  $\delta$  be the dyscope of the last frame on  $\text{stack}(p)$ ;
4   let  $\sigma = \text{static}(\delta)$ ;
5   let  $\sigma' = \text{lscope}(l')$ ;
6   let  $\sigma_{\text{lub}}$  be the least upper bound of  $\sigma$  and  $\sigma'$  in the tree  $\Sigma$ ;
7   let  $m$  be the minimum integer such that  $\text{sparent}^m(\sigma) = \sigma_{\text{lub}}$ ;
8   let  $\delta_{\text{lub}} = \text{dparent}^m(\delta)$ ;
9   let  $n$  be the minimum integer such that  $\text{sparent}^n(\sigma') = \sigma_{\text{lub}}$ ;
10  let  $\delta_0, \dots, \delta_{n-1}$  be  $n$  distinct objects not in  $\Delta$ ;
11  let  $\Delta' = \Delta \cup \{\delta_0, \dots, \delta_{n-1}\}$ ;
12  define  $\text{dparent}' : \Delta' \setminus \{\text{droot}\} \rightarrow \Delta'$  by
      
$$\text{dparent}'(\delta') = \begin{cases} \text{dparent}(\delta') & \text{if } \delta' \in \Delta \\ \delta_{i+1} & \text{if } \delta' = \delta_i \text{ for some } 0 \leq i < n-1; \\ \delta_{\text{lub}} & \text{if } n \geq 1 \text{ and } \delta' = \delta_{n-1} \end{cases}$$

13  define  $\text{static}' : \Delta' \rightarrow \Sigma$  by  $\text{static}'(\delta') = \begin{cases} \text{static}(\delta') & \text{if } \delta' \in \Delta \\ \text{sparent}^i(\sigma') & \text{if } \delta' = \delta_i \text{ for some } 0 \leq i < n \end{cases}$ ;
14  for  $\delta' \in \Delta'$  and  $v \in \text{vars}(\text{static}'(\delta'))$ , let  $\text{deval}'(\delta')(v) = \begin{cases} \text{deval}(\delta')(v) & \text{if } \delta' \in \Delta \\ \text{default}_t & \text{otherwise} \end{cases}$ ;
15  define  $\text{stack}'$  to be the same as  $\text{stack}$  except that the last frame on  $\text{stack}'(p)$  is replaced with
       $(\delta_0, l')$  if  $n \geq 1$ , or with  $(\delta_{\text{lub}}, l')$  if  $n = 0$ ;
16  let  $s' = (\Delta', \text{droot}, \text{dparent}', \text{static}', \text{deval}', P, \text{stack}')$ ;
17  return the result of removing all unreachable dyscopes from  $s'$ ;

```

Figure 12.1: Jump protocol: how the state changes when control moves to a new location within a function. The procedure may only be called if $\text{func}(\sigma) = \text{func}(\sigma')$, i.e., the jump is contained within one function.

idea is that you have to move up the dyscope tree every time you move past a right curly brace (i.e., leave a scope) and then push on a new scope for each left curly brace you move past. So there is a sequence of upward moves followed by a sequence of pushes to get to the new location. (And either or both sequences could be empty.) At the end, if any dyscopes become unreachable, they are removed from the state.

Note however, that we do not assume that scopes are associated to locations in a nice lexical pattern (or any pattern at all). The protocol described here works for any arbitrary assignment of scopes to locations.

The precise protocol is described in Figure 12.1. The algorithm shown there takes as input a well-formed state, a process ID, and a location l' for the function that p is currently in. Say the current dyscope for p is δ , and l' is in static scope σ' . Let $\sigma = \text{static}(\delta)$. Hence the current static scope is σ and the new static scope will be σ' .

First, you have to find the *least upper bound* σ_{lub} of σ and σ' in the static scope tree. (Hence σ_{lub} is a common ancestor of σ and σ' , and if σ'' is any common ancestor of σ and σ' then σ'' is an ancestor or equal to σ_{lub} .) Note that the least upper bound must exist since the function scope is a common ancestor of σ and σ' . There is a chain of static scopes from σ up to σ_{lub} and a corresponding chain $\delta, \text{dparent}(\delta), \dots, \text{dparent}^m(\delta)$ in the dynamic scope tree. Let $\delta_{\text{lub}} = \text{dparent}^m(\delta)$. This will

become the least upper bound of δ and the new dynamic scope corresponding to σ' that will be added to the state.

Next you add new dyscopes corresponding to the chain of static scopes leading from σ_{lub} down to σ' . The variables in the new scopes are assigned the default values for their respective types. The `dparent`, `static`, and `deval` maps are adjusted accordingly. Finally, the stack is modified by replacing the last activation frame with a frame referring to the (possibly) new dynamic scope and new location l' .

This protocol is executed every time control moves from one location to another.

Note that in CIVL all jumps stay within a function. There is no way to jump from one function to another (without returning).

A small variation is the protocol for moving to the start location of a function when the function is first pushed on the stack. Since the start location is not necessarily in the function scope (it may be a proper descendant of that scope), you have to execute the second half of the protocol to push a sequence of scopes from the function scope to the scope of the start location.

12.3 Initial State

The *initial state* for M is obtained by creating one process (let $P = \{0\}$) and having it call the root function f_0 . Hence start with the state s in which $P = \{0\}, \dots$. The initial state is `jump( $s, 0, \text{start}_{f_0}$ )`.

12.4 Transitions

The transitions follow the usual “interleaving” semantics. Given a state σ , one defines the set of enabled transitions in σ as follows. Let $p \in P$. Look at the last frame (d, l) of `stack( $p$ )` (i.e., the top of the call stack), assuming the stack is nonempty. Look at all guarded transitions emanating from l . For each such transition, evaluate the guard using the valuation formed by taking the union of the valuations of all ancestors of d (including d). In other words, follow the standard “lexical scoping” protocol to determine the value of any variable that could occur at this point. Those transitions whose guard evaluates to *true* are enabled.

For each enabled transition, a new state is generated by executing the transition’s statement. For the most part, the semantics are obvious, but there are a few details that are a bit subtle.

12.5 Calls and Spawns

Both calls and forks of a function f entail the creation of a new frame. First, a new dyscope d corresponding to `fscope( $f$ )` is created. To find out where to stick that new scope in the dynamic scope tree, proceed as follows: begin in the dyscope for the process invoking the fork or call and start moving up its parent sequence until you reach the first dyscope d' whose associated static scope is the defining scope of f . (You must reach such a scope, or else f would not be visible, and the model would have a syntax error!) Insert d right under d' , i.e., `dparent( $d$ ) =  $d'$` . This preserves the required correspondence between static scopes and dyscopes. Now you move to the start location, using the jump protocol, which may involve the creation of additional dyscopes under d . The new frame references the last dynamic scope you created, and the location is the start location of f . Variables are given their initial values in all the newly created dyscopes (however that is done).

All of that is the same whether the statement is a fork or call. The only difference is what happens next: for a call, the new frame is pushed onto the calling process' call stack. For a fork, a new process is “created”, i.e., you pick a natural number not in P and add it to P , and push the frame onto the new stack. To be totally deterministic, you could pick the least natural number not in P .

12.6 Garbage collection

In a state s , a dyscope is unreachable if there is no path from a frame in a call stack to that dyscope (following the **dparent** edges). You can remove all unreachable dyscopes.

If a process has terminated (has empty stack) and *there are no references to that process* in the state, you can just remove the process from the state.

In any state, you can renumber the processes (and update the references accordingly) however you want, to get rid of gaps, put them in a canonic order, etc.

Part IV

Tools

Chapter 13

Tool Overview

13.1 Symbolic execution

The tools currently in the CIVL tool kit all use *symbolic execution*. This is a technique in which variables are assigned symbolic rather than concrete values. In particular, input variables are assigned unique *symbolic constants*, which are symbols such as X , Y , and so on. Operations produce symbolic expressions in those symbols, such as $(X + Y)/2$.

13.2 Commands

Current tools allow one to *run* a CIVL program using random choice to resolve nondeterministic choices; *verify* a program using model checking to explore all states of the program; and *replay* a trace if an error is found. There are also commands to show the results just of preprocessing or parsing a file; as these are basically sub-tasks of the other tasks, they are used mainly for debugging.

Each tool is launched from the command line by typing “`civl cmd ...`”, where *cmd* is one of

- `help` : print usage information
- `run` : run the program using random simulation
- `verify` : verify program
- `replay` : replay trace for program
- `parse` : show result of preprocessing and parsing file
- `preprocess` : show result of preprocessing file only.

The additional arguments and options are described below and are also shown by the `help` command.

A number of properties are checked when running or verifying a CIVL program. These include the following:

- absence of deadlocks
- absence of assertion violations
- absence of division or modulus with denominator 0

- absence of illegal pointer dereferences
- absence of out-of-bounds array indexes
- absence of invalid casts
- every object is defined (i.e., initialized) before it is used.

13.3 Options

The following command line options are available:

- `-debug` or `-debug=boolean` (default: `false`)
debug mode: print very detailed information
- `-echo` or `-echo=boolean` (default: `false`)
print the command line
- `-enablePrintf` or `-enablePrintf=boolean` (default: `true`)
enable `printf` function
- `-errorBound=integer` (default: 1)
stop after finding this many errors
- `-guided` or `-guided=boolean` (default: `false`)
user guided simulation; applies only to `run`, ignored for all other commands
- `-id=integer` (default: 0)
ID number of trace to replay
- `-inputkey=value`
initialize input variable *key* to *value*
- `-maxdepth=integer` (default: 2147483647)
bound on search depth
- `-min` or `-min=boolean` (default: `false`)
search for minimal counterexample
- `-por=string` (default: `std`)
partial order reduction (por) choices: `std` (standard por) or `scp` (scoped por)
- `-random` or `-random=boolean` (default: `varies`)
select enabled transitions randomly; default for `run`, ignored for all other commands
- `-saveStates` or `-saveStates=boolean` (default: `true`)
save states during depth-first search
- `-seed=string` (default: `none`)
set the random seed; applies only to `run`
- `-showAmpleSet` or `-showAmpleSet=boolean` (default: `false`)
print the ample set of each state

`-showModel` or `-showModel=boolean` (default: `false`)
 print the model

`-showProverQueries` or `-showProverQueries=boolean` (default: `false`)
 print theorem prover queries only

`-showQueries` or `-showQueries=boolean` (default: `false`)
 print all queries

`-showSavedStates` or `-showSavedStates=boolean` (default: `false`)
 print saved states only

`-showStates` or `-showStates=boolean` (default: `false`)
 print all states

`-showTransitions` or `-showTransitions=boolean` (default: `false`)
 print transitions

`-simplify` or `-simplify=boolean` (default: `true`)
 simplify states?

`-solve` or `-solve=boolean` (default: `false`)
 try to solve for concrete counterexample

`-sysIncludePath=string` (default:)
 set the system include path

`-trace=string` (default:)
 filename of trace to replay

`-userIncludePath=string` (default:)
 set the user include path

`-verbose` or `-verbose=boolean` (default: `false`)
 verbose mode

13.4 Errors

When a property violation occurs, either in *verification* or *run* mode, a brief report is written to the screen. In addition, a report may be *logged* in the directory `CIVLREP`.

The *error bound* parameter determines how many errors can be encountered before a search terminates. By default, the error bound is 1, meaning a search will stop as soon as the first error is found. The error bound can be set to a higher number on the command line using option *errorBound*.

When the error bound is greater than one, the CIVL verifier continues searching after the first error is discovered. It first attempts to “recover” from the error by adding to the path condition a clause which guarantees that the error cannot happen. For example, if the error was caused by a possible division by zero, x/y , where y is an unconstrained real symbolic constant, CIVL will add to the path condition the predicate $y \neq 0$, and continue the search. In some cases, CIVL determines that the modified path condition is unsatisfiable, in which case the search will back-track in the usual symbolic execution way.

In addition to the printed reports, errors are logged. However, CIVL follows a protocol aimed at limiting the number of reports of errors which are essentially the same. This protocol uses a simple equivalence relation on the set of errors. Two erroneous states are considered equivalent if the errors are of the same *kind* (deadlock, division by zero, illegal pointer dereference, etc.) and every process is at the same location in both states. When an error is encountered, CIVL first checks to see if an earlier equivalent errors exists in the log. If so, the lengths of the traces leading to the two error states are compared. If the new trace is shorter, the old log entry is replaced with the new one. In this way, only the shortest representative error trace for each equivalence class of errors is recorded in the log.

A log entry actually entails two things: first, a plain text entry similar to the one printed to the screen is made in a log file in CIVLREP. The name of this file is usually of the form *root_log.txt*, where *root* is the root of the original file name, i.e., the substring of the file name that ends just before the first ‘.’. For example, if the file name is *diningBad.cvl*, the log file will be named *diningBad_log.txt*. This is a plain text, human-readable file which summarizes the results of the verification run.

In addition, each saved trace is stored in a separate file in CIVLREP. The names of these files have the form *root_id.trace*, where *id* is the error ID (reported when the error is printed and logged). This file is not intended to be human-readable. It contains a compressed representation of the trace, including all of the options and parameter values and choices made a nondeterministic points. This file is used by CIVL when replaying an error trace.

As mentioned above, the CIVL **replay** command is used to play an earlier-saved error trace. When more than one trace has been saved, the **-id** command line option can be used to specify which one to play. (The default *id* is 0).

Chapter 14

Interpreting the Output

14.1 Transitions

Transitions are printed during trace `replay`, in the course of a `run`, or during verification if option `showTransitions` is selected. A typical transition is printed as follows:

```
State 6, proc 0:
  44->49: LOOP_FALSE_BRANCH at f0:36.18-23 "i < n";
  49->RET: return (init) at f0:37.0-1 "}";
  7->8: i = 0 at f0:42.7-16 "int i = 0";
--> State 74
```

This means that the transition begins executing from the state with ID 6, and it is executed by the process with PID 0. The transition is executed in a sequence of atomic steps; in this case there the transition consists of three steps.

Process 0 begins at location 44; this is a static location in the program graph of a function in the CIVL model. Details about the locations can be seen by printing the CIVL model. In executing the first step, control moves from location 44 to location 49. This first step is an edge in the program graph corresponding to the *false* branch of a loop condition, i.e., the branch that exits the loop.

The remainder of the line describing the step specifies the part of the original source code corresponding to this step. The source code fragment can be found in file `f0`. To save space and avoid constantly repeating long paths, all the source files involved in a program are printed once and assigned keys such as `f0`, `f1`, etc. The legend is printed out once at the beginning of the run; in this case it is simply the following:

```
File name list:
f0 : dining.cvl
```

The source code fragment begins on character 18 of line 36 of `f0`, and extends to character 23 of that line. This range is inclusive on the left and exclusive on the right, so the total number of characters in this range is $23 - 18 = 5$. The five characters from the source code are printed next inside double quotes. For longer ranges, this excerpt will be abbreviated using an elipsis.

The second step executes a *return* statement, which results in popping the top activation frame from process 0's call stack. The function returning is `init`. Since the program counter for that frame disappears with the execution of this step, there is no final value for its new location; this is signified using the pseudo-location `RET`.

In the new top frame, control is at location 7, and an assignment statement is executed, moving control to location 8. With this last step, the transition ends at State 74.

Between transitions, processes can be renumbered. Hence a process with PID 0 in one state, may have a different PID in another state. The same is true for dynamic scope IDs. Within a single transition, however, these numbers will not change.

14.2 States

States are printed typically when a property is violated, at the initial or final points of a trace replay, or if the option *showStates*, *showSavedStates*, *verbose*, or *debug* is selected.

A complete print-out of a state can be seen in Figure 14.1. The state is presented in hierarchical way. At the top-most level of this hierarchy, there are 3 main parts to the state:

1. the *path condition*, i.e., the boolean-valued symbolic expression used in symbolic execution to keep track of all conditions on the input symbols which must hold in order for the current path to have been followed;
2. the *dynamic scopes*, and
3. the *process states*.

The dynamic scopes are numbered starting from 0. This numbering is arbitrary and is invisible to the program, i.e., there is no way for the program to obtain its dynamic scope ID. This allows the verifier to renumber dynamic scopes at will in order to transform a state into an equivalence canonical form.

The print-out of each dynamic scope specifies the ID of the dyscope's parent in the dyscope tree. (The root dyscope shows -1 for the parent.) This specifies the complete tree structure of the dyscopes. Each dyscope is an instance of some static scope; the representation also shows the ID of this static scope.

The next line in the representation of the dyscope shows a set of *reachers*. These are the PIDs of the processes that can *reach* this dyscope. A process can reach a dyscope if there is path in the dyscope tree that starts from a dyscope referenced by a frame on the process' call stack and follows the parent edges in the tree.

The *variables* section of the dyscope representation consists of one line for each variable in the static scope corresponding to the dyscope. There are also special hidden variables, such as the heap. In each case, the value assigned to the variable is shown. A value of `NULL` indicates that the variable is currently undefined.

The process states section consists of one sub-section for each process currently in the state. Like the dynamic scopes, the processes are numbered in some arbitrary way. For each process, the value of the *atomic count* is given. This is the nesting depth of the atomic blocks in which the process is currently located, i.e., the number of times the process has entered an atomic block without exiting the block.

The call stack of the process lists the activation frames on the stack from top to bottom. The frame at the top correspond to the function currently executing in that process. The name of the function, the value of the program counter (location), and the source code for that location, and the dyscope ID for the frame are shown.

```

State 84
| Path condition
| | true
| Dynamic scopes
| | dyscope 0 (parent=-1, static=0)
| | | reachers = {1,2}
| | | variables
| | | | __atomic_lock_var = process<-1>
| | | | __heap = NULL
| | | | BOUND = 3
| | | | n = 2
| | | | forks = X_s0v4[0:=0, 1:=-1]
| | dyscope 1 (parent=2, static=5)
| | | reachers = {1}
| | | variables
| | | | __heap = NULL
| | | | left = 0
| | | | right = 1
| | dyscope 2 (parent=0, static=1)
| | | reachers = {1}
| | | variables
| | | | id = 0
| | | | __heap = NULL
| | dyscope 3 (parent=4, static=5)
| | | reachers = {2}
| | | variables
| | | | __heap = NULL
| | | | left = 1
| | | | right = 0
| | dyscope 4 (parent=0, static=1)
| | | reachers = {2}
| | | variables
| | | | id = 1
| | | | __heap = NULL
| Process states
| | process 0
| | | atomicCount = 0
| | | call stack
| | process 1
| | | atomicCount = 0
| | | call stack
| | | | Frame[function=dine, location=25, f0:22.32-44 "forks[right]", scope=1]
| | process 2
| | | atomicCount = 0
| | | call stack
| | | | Frame[function=dine, location=33, f0:26.32-44 "forks[right]", scope=3]

```

Figure 14.1: Complete print-out of a state

14.3 Property Violations

As described in Section 13.4, an error report is printed whenever CIVL encounters an error. A typical error report appears as follows:

```
Error 0 encountered at depth 21:
CIVL execution error (kind: DEADLOCK, certainty: PROVEABLE)
A deadlock is possible:
  Path condition: true
  Enabling predicate: false
ProcessState 0: terminated
ProcessState 1: at location 25, f0:21.30-42 "forks[right]"
  Enabling predicate: false
ProcessState 2: at location 25, f0:21.30-42 "forks[right]"
  Enabling predicate: false
at f0:21.30-42 "forks[right]".
```

The report begins with “Error 0”. The errors are numbered in the order they are discovered in this search; this indicates that this is the first (0th) error encountered. The depth refers to the length of the depth-first search stack when the error was encountered. In this case, the depth was 21, meaning that the trace leading to the erroneous state is a sequence of 21 states and 20 transitions.

The errors are categorized by *kind*. The error kinds include *deadlock*, indicating that it is possible no transition is enabled in the state; *assertion violation*, indicating an assertion may fail in the state; *division by zero*; and *out of bounds*, indicating an array index may be out of bounds, among many more.

In addition to the brief report shown above, most error reports also include a complete print-out of the state at which the error occurred. They will also include additional information specific to the kind of error. For example, the deadlock error report shown above includes the following information:

- the value of the path condition;
- the *enabling predicate*, which is the disjunction of the guards associated to all transitions departing from the current state. This is the predicate that CIVL has found to possibly be unsatisfiable under the context of the path condition; and
- for each process, the current location of the process and the enabling predicate for that process, i.e., the disjunction of the guards associated to all transitions departing from the current state in that process (CIVL has found that all of these may be unsatisfiable).

Errors are also categorized as to their *certainty*. CIVL is *conservative*, meaning that if it not sure a property holds in a state, it will report it. This means that it may sometimes raise *false alarms*, i.e., report a possible error even when none exists. The certainty measures how certain CIVL is that the error is real. The certainty levels, from most to least certain, are as follows:

1. *concrete*: this indicates that CIVL has actually found concrete values for all input variables that are guaranteed to drive the execution along the current trace and result in the error;
2. *proveable*: this indicates that a theorem prover (either the external one or CIVL’s built-in prover) has determined that the error is feasible, which includes proving that the path condition is satisfiable; however, it has not necessarily found concrete values for the inputs;

3. *maybe*: this indicates the prover is not sure whether this is an error; this could be due to the incompleteness of the decision procedure, or it could be a real error;
4. *none*: probably an internal CIVL error: the theorem prover has not said anything.

14.4 Statistics

- *validCalls*: the number of calls to the CIVL *valid* method, used to determine if a first-order formula is valid under a given first-order context. Some of these queries are resolved quickly by CIVL; when CIVL cannot resolve the query itself, it calls a separate theorem prover (CVC3)
- *proverCalls*: the number of calls to the separate theorem prover's *valid* method
- *memory*: the total amount of memory, in bytes, consumed by the Java virtual machine at the end of the search/run.
- *time*: the total time, in seconds, used to perform the CIVL operation
- *maxProcs*: the maximum process count achieved, over all states encountered in the search/run
- *statesInstantiated*: the number of state objects instantiated during the course of the verification/run
- *statesSaved*: the number of states saved in the course of a search
- *statesSeen*: the number of states pushed onto the depth-first search stack in the course of the search; note that “intermediate states” created in the process of executing a transition are not pushed onto the stack, only the final state resulting from the transition is pushed
- *statesMatched*: the number of times a state encountered in the depth-first search was found to match a saved state seen earlier in the search
- *steps*: the total number of primitive steps executed during the verification/run. A step is the smallest, atomic, unit of execution; each transition is composed of one or more steps. This number is a good measure for the total amount of “work” carried out by CIVL
- *transitions*: the total number of transitions executed during the verification/run. A transition is a coarser unit of execution; each transition consists of one or more steps executed from a single process, resulting in a state which is then pushed onto the DFS stack.

Chapter 15

Emacs mode

A CIVL-C mode for the **emacs** text editor is available in directory **emacs** in the CIVL distribution. This provides syntax highlighting and automatic indentation for CIVL-C programs.

To install this mode:

1. Copy file `civl-syntax.el` to `~/.emacs.d/lisp` or another favorite location
2. Include that path in your load path in `.emacs`:

```
(add-to-list 'load-path "~/.emacs.d/lisp")
```

3. Add the following lines to your `~/.emacs` file:

```
(require 'civl-syntax)
(civl-syntax)
```

We are grateful to William Killian of the University of Delaware for writing this emacs module.

Bibliography

- [1] KHRONOS GROUP. OpenCL: The open standard for parallel programming of heterogeneous systems. <https://www.khronos.org/opencv/>.
- [2] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard, version 3.0. <http://www.mpi-forum.org/docs/docs.html>, Sept. 2012.
- [3] NVIDIA. CUDA C Programming Guide Version 5.0. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>. accessed March 3, 2014.
- [4] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP API Specification for Parallel Programming. <http://openmp.org/wp/>. accessed Feb. 8, 2014.