

**CIVL**  
The Concurrency Intermediate Verification Language  
Reference Manual  
v0.5

Matthew B. Dwyer  
Stephen F. Siegel

Ganesh Gopalakrishnan  
Manchun Zheng

Zvonimir Rakamaric  
Timothy K. Zirkel

February 8, 2014

# Contents

<b>1</b>	<b>Quick Start</b>	<b>2</b>
<b>I</b>	<b>Language</b>	<b>3</b>
<b>2</b>	<b>Overview of CIVL-C</b>	<b>4</b>
2.1	CIVL-C concepts and primitives . . . . .	4
2.2	Detailed descriptions of primitives . . . . .	4
2.2.1	\$proc . . . . .	4
2.2.2	\$assert . . . . .	5
2.2.3	\$assume . . . . .	5
2.2.4	\$atom . . . . .	6
2.2.5	\$atomic . . . . .	6
2.2.6	\$choose . . . . .	7
2.2.7	\$choose_int . . . . .	7
2.2.8	\$input . . . . .	7
2.2.9	\$invariant . . . . .	8
2.2.10	\$output . . . . .	8
2.2.11	\$self . . . . .	8
2.2.12	\$spawn . . . . .	8
2.2.13	\$wait . . . . .	8
2.2.14	\$when . . . . .	8
2.2.15	Procedure contracts . . . . .	9
2.2.16	Remote expressions . . . . .	10
2.2.17	Collective expressions . . . . .	10
<b>3</b>	<b>Pointers and heaps</b>	<b>11</b>
3.1	Pointer types . . . . .	11
3.2	Address-of operator . . . . .	11
3.3	Pointer addition and subtractions . . . . .	12
3.4	Semantics of scopes and pointer types . . . . .	12
3.5	Pointer casts . . . . .	12
3.6	Heaps . . . . .	13
3.7	Scope-Parameterized Functions . . . . .	13
3.8	Scope-Parameterized Type Definitions . . . . .	14

II

Semantics

15

4

CIVL Model Syntax

16

4.1	Notation and terminology . . . . .	16
4.2	Definition of Context . . . . .	16
4.3	Lexical scopes . . . . .	17
4.4	Functions . . . . .	17
4.5	Statements . . . . .	20
4.6	Remarks . . . . .	21
4.7	Transition system representation of functions . . . . .	22

5

CIVL Model Semantics

23

5.1	State . . . . .	23
5.2	Jump protocol . . . . .	24
5.3	Initial State . . . . .	26
5.4	Transitions . . . . .	26
5.5	Calls and Spawns . . . . .	26
5.6	Garbage collection . . . . .	27

III

Tools

28

6

Overview of CIVL Tools

29

7

Transitions

31

# Chapter 1

## Quick Start

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.
2. Move the resulting directory `vsl` to `/opt/vsl`.
3. 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.
4. 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`.
5. You should now have an executable script `/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 `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 2 processes try to obtain the locks in opposite order, leading to a deadlock.

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.

**Note.** You can install `CIVL-tag` and `vsl` in any directory you want, not just in `/opt`. You just need to edit the script file `civl` appropriately, replacing the default paths with the new paths.

**Part I**

**Language**

## Chapter 2

# Overview of CIVL-C

### 2.1 CIVL-C concepts and primitives

CIVL-C is a programming language. It is an extension of a subset of the C11 dialect of C. It does not include the standard C library.

A CIVL-C program should begin with the line

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

which includes the main CIVL-C header file, which declares all the types and other CIVL primitives. Almost all of the CIVL-C primitives which are not already in C begin with the symbol `$` to easily distinguish them from any reserved work or identifier in a C program.

A CIVL-C program encodes a CIVL model for a particular CIVL context. The types are essentially the C types with the additional process reference type (denoted `$proc`). The expressions are C expressions with some additional expressions defined below.

In CIVL-C, functions can be defined in any scope, not just in file scope. The lexical scope structure and placement of function definitions determine the static scope tree  $\Sigma$  and the function prototype system. A function’s defining scope is, as you would expect, the scope in which its definition occurs.

The CIVL-C code will not have an explicit “root” procedure. Instead, a root procedure will be implicitly wrapped around the entire code. The global input variables will become the inputs to the root procedure. A “`main`” procedure must be declared that takes no parameters but can have any return type. The body of `main` becomes the body of the root procedure. The return type of `main` becomes the return type of the root procedure. The `main` procedure itself disappears in translation.

The reason for this protocol is that an arbitrary (sequential) C program is a legal (and reasonable) CIVL-C program. The global variables in the C program simply become variables declared in the root scope.

The additional language elements are shown in Figure 2.1.

### 2.2 Detailed descriptions of primitives

#### 2.2.1 `$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

<code>\$assert</code>	check something holds
<code>\$assume</code>	assume something holds
<code>\$atom</code>	defines statements to be executed as one transition
<code>\$atomic</code>	defines statements to be executed without interleaving other processes
<code>\$choose</code>	nondeterministic choice statement
<code>\$choose_int</code>	nondeterministic choice of integer
<code>\$collective</code>	a collective expression
<code>\$ensures</code>	procedure postcondition
<code>\$false</code>	boolean value false, used in assertions
<code>\$heap</code>	the heap type
<code>\$input</code>	type qualifier declaring variable to be a program input
<code>\$invariant</code>	declare a loop invariant
<code>\$malloc</code>	malloc function with additional heap arguments
<code>\$output</code>	type qualifier declaring variable to be a program output
<code>\$proc</code>	the process type
<code>\$requires</code>	procedure precondition
<code>\$result</code>	refers to result returned by procedure in contracts
<code>\$scope</code>	the scope type, used to give a name to a scope
<code>\$self</code>	the evaluating process (constant of type <code>\$proc</code> )
<code>\$spawn</code>	create a new process running procedure
<code>\$true</code>	boolean value true, used in assertions
<code>\$wait</code>	wait for a process to terminate
<code>\$when</code>	guarded statement
<code>@</code>	refer to variable in other process, e.g., <code>p@x</code>
<code>*&lt;...&gt;</code>	scope-qualified pointer type

Figure 2.1: CIVL-C primitives. Some of these are part of the grammar of the language; others are defined in the header file `civlc.h`.

and cannot be cast to one. Certain expressions take an argument of `$proc` type and some return something of `$proc` type.

### 2.2.2 `$assert`

This is an assertion statement. It takes as its sole argument an expression of boolean type. The expressions have a richer syntax than C expressions. During verification, the assertion is checked. If it does not hold, a violation is reported.

```
$assert expr;
```

Boolean values `$true` and `$false` may be used in assertions and assumptions.

### 2.2.3 `$assume`

This is an assume statement. Its syntax is the same as that of `$assert`. 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.

```
$assume expr;
```

### 2.2.4 `$atom`

This defines a number of statements to be executed as one 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 is deterministic, non-blocking and finite. If one of the requirement is found to be violated, an error or a warning will be reported. For example, an error will be reported when executing the code below, because the `$wait` statement is blocked.

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

### 2.2.5 `$atomic`

This defines a number of statements that will only allow the process to interleave with others when necessary. An `$atomic` block has the following form.

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

A process executing an `$atomic` block will try to execute statements contained in the `$atomic` block without interleaving with other processes, unless the process is blocked. For example, the following loop will be executed without interleaving with other processes.

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

When no transition can be enabled, the execution of the `$atomic` block will be interrupted. And other processes are allowed to execute, until the process gets enabled again to continue executing its `$atomic` block. For example, in the following lines of code, after executing the first loop, the `$atomic` block is blocked, because it has to wait for other processes' termination.

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



### 2.2.6 \$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:

```
l1: $choose {
    $when (x>0) {x--; goto l2;}
    $when (x==0) {y=1; goto l3;}
    default: {z=1; goto l4;}
}
l2: $choose {
    ...
}
l3: $choose {
    ...
}
```

### 2.2.7 \$choose\_int

This is a function with the following prototype:

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

It takes as input a positive integer `n` and non-deterministically returns an integer in the range  $[0, n - 1]$ .

### 2.2.8 \$input

A variable in the root scope only may be declared with this type modifier indicating it is an “input” variable, as in

```
$input int n;
```

As explained above, the variable becomes a parameter to the root procedure. This is used when comparing two programs for functional equivalence. The two programs are 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). Input variables can also be assigned a concrete value on the command line.

### 2.2.9 \$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.

### 2.2.10 \$output

A variable in the root scope may be declared with this type modifier to declare it to be an output variable.

### 2.2.11 \$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`”.

### 2.2.12 \$spawn

This 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.

### 2.2.13 \$wait

This is a statement that takes an argument of type `$proc` and blocks until the referenced process terminates:

```
$wait expr;
```

### 2.2.14 \$when

This represents a guarded command:

```
$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;
```

### 2.2.15 Procedure contracts

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.

### 2.2.16 Remote expressions

. 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.

### 2.2.17 Collective expressions

. 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 3

# Pointers and heaps

CIVL-C supports pointers, using the same operators with the same meanings as C (`&`, `*`, pointer arithmetic). As mentioned above, there is also a heap type `$heap`, which can be used to declare multiple heaps in a CIVL-C program. The interaction between pointers, heaps, and scopes is an important aspect of CIVL-C.

### 3.1 Pointer types

Given any object type  $T$  and a static scope  $s$  in a CIVL-C program, there is a type *pointer-to- $T$ -in- $s$* . The type is used to represent a pointer to a memory location of type  $T$  in scope  $s$  or a descendant of  $s$  (i.e., some scope contained in  $s$ ).

If scope  $s_1$  is a descendant of  $s_2$  (i.e.,  $s_1$  is lexically contained in  $s_2$ ), the type *pointer-to- $T$ -in- $s_1$*  is a subtype of *pointer-to- $T$ -in- $s_2$* . This means that any expression of the first type can be used wherever an object of the second type is expected. In particular, any expression  $e$  of the subtype can be assigned to a left-hand-side expression of the supertype without explicit casts; also  $e$  can be used as an argument to a function for which the corresponding parameter has the supertype.

The syntax for denoting this type adheres to the usual C syntax for denoting the type *pointer-to- $T$*  with the addition of a scope parameter within angular brackets immediately following the `*` token. For example, to declare a variable  $p$  of type *pointer-to- $T$ -in- $s$* , one writes

```
int *<s> p;
```

If the scope modifier `<...>` is absent, the scope is taken to be the root scope  $s_0$ . The object has type *pointer-to- $T$ -in- $s_0$* , which is abbreviated as *pointer-to- $T$* . In this way, standard C programs can be interpreted as CIVL-C programs.

### 3.2 Address-of operator

The address-of operator `&` returns a pointer of the appropriate subtype using the innermost scope in which its left-hand-side argument is declared. For example

```
{
    $scope s1;
    int x;
    double a[N];
    int *<s1> p = &x;
```

```

    double *<s1> q = &a[2];
}

```

is correct (in particular, it is type-correct) because  $\&x$  has type *pointer-to- $\mathbf{int}$ -in- $s_1$* , since  $s_1$  is the scope in which  $x$  is declared.

### 3.3 Pointer addition and subtractions

If  $e$  is an expression of type *pointer-to- $T$ -in- $s$*  and  $i$  is an expression of integer type then  $e+i$  also has type *pointer-to- $T$ -in- $s$* . In other words, pointer addition cannot leave the scope of the original pointer. This reflects the fact that every object is contained in one scope, and pointer addition cannot leave the object.

Pointer subtraction is defined on two pointers of the same type, where “same” includes the scope. That is checked statically. As in C, it is only defined if the two pointers point to the same object. In CIVL-C, a runtime error will be thrown if they do not point to the same object.

### 3.4 Semantics of scopes and pointer types

A variable of type  $\$scope$  is treated like any other variable. It becomes part of the state when the scope in which it is declared is instantiated to form a dynamic scope. The variable is initialized at that time and its value cannot change.

Each time a dynamic scope is instantiated, it is assigned a unique ID number. The exact value of the ID number is not relevant, it just has to be distinct from any other scope ID number that currently exists in the state. This is the value that is assigned to the scope variable. Therefore, if a static scope contains a scope variable, and that scope is instantiated twice to form two distinct dynamic scopes, the values assigned to the two variables will be distinct.

A pointer value is an ordered pair  $\langle \delta, r \rangle$ , where  $\delta$  is a dynamic scope ID and  $r$  is a reference to a memory location in the static scope associated to  $\delta$ . (We will define the exact form of a reference later.)

When a dynamic scope is instantiated, each new variable created is assigned a *dynamic type*. This is a refinement of the static type associated to the static variable. Every dynamic type is an instance of exactly one static type. The dynamic type of the newly instantiated variable is an instance of the static type of the static variable.

The dynamic pointer types have the form *pointer-to- $t$ -in- $\delta$* , where  $t$  is a dynamic type and  $\delta$  is a dynamic scope ID. For a program to be dynamically type safe, such a variable should hold only values of the form  $\langle \delta, r \rangle$ . In particular, the variable should never be assigned a value where the dynamic scope component is a different instance of the static scope  $s$  associated to  $\delta$ .

### 3.5 Pointer casts

If scope  $s_1$  is contained in scope  $s_2$ , an expression of type *pointer-to- $T$ -in- $s_1$*  can always be cast to *pointer-to- $T$ -in- $s_2$* , because the first is a subtype of the second. (As described above, the cast is unnecessary.)

The cast in the other direction is also allowed, but the dynamic type safety of that cast will only be checked at runtime. In particular, a runtime error will result if the cast attempts to cast the pointer value to a dynamic scope which does not contain (is an ancestor of) the dynamic scope component of the pointer value.

A type *pointer-to- $T_1$ -in- $s$*  can be cast to a type *pointer-to- $T_2$ -in- $s$*  according to the usual rules of C. In other words, usual casting rules apply as long as you don't change the scope.

## 3.6 Heaps

The standard CIVL-C library defines a type `$heap` for explicit modeling of a heap. The default value of `$heap` type is an empty heap, so you only need to declare a variable to have type `$heap` in order to create a new heap:

```
$heap h; /* a new empty heap */
```

The following functions are also defined:

```
void* $malloc($heap *h, int size);
void memcpy(void *p, void *q, size_t size);
void free(void *p)
```

The first function is like C's `malloc`, except that you specify the heap in which the allocation takes place by passing a pointer to the heap as the first argument. This modifies the specified heap and returns a pointer to the new object. The function can only occur in a context in which the type of the object is specified, as in:

```
$heap h;
int n = 10;
double *p = (double*)$malloc(&h, n*sizeof(double));
```

The second and third functions are exactly the same as in C. Note that `free` modifies the heap which was used to allocate `p`.

Another pointer example:

```
{ $heap h;
  { $scope s1;
    double x;
    { $scope s2;
      double y;
      double *<s1> p;
      /* p can only point to something in s1 or descendant,
       * for example, s2 */
      p = &x; // fine
      p = &y; // fine
      p = (double*)$malloc(&h,10*sizeof(double)); // static type error
    }
  }
}
```

*Status:* Pointer type, `&`, `*`, pointer addition all implemented. Type `$heap`, `malloc`, `free` implemented. Scope-qualified pointers: parsed, type-checked, but information not currently used.

## 3.7 Scope-Parameterized Functions

Coming soon. (Parsed, type checked, not currently used otherwise.)

## 3.8 Scope-Parameterized Type Definitions

Coming soon. (Ditto.)



# Part II

## Semantics

# Chapter 4

## CIVL Model Syntax

### 4.1 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  $\text{length}(\xi)$ .

### 4.2 Definition of Context

**Definition 4.2.0.1.** A *CIVL type system* is a tuple comprising the following components:

1. a set  $\text{Type}$  (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  $\text{Var}$  (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  $\text{Val}$  (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 4.2.0.2.** 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  $\text{Eval}$  denote the set of all valuations in that system.

**Definition 4.2.0.3.** Given a CIVL type system, A *CIVL expression system* for that type system is a tuple comprising the following components:

1. a set  $\text{Expr}$  (the set of all *typed expressions* over  $\text{Var}$ ),
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 4.2.0.4.** A *CIVL context* is a CIVL type system together with a CIVL expression system for that type system.

### 4.3 Lexical scopes

**Definition 4.3.0.5.** 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  $\Sigma$  (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  $\Sigma$  the structure of a rooted tree with root  $\text{root}$ ,

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

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

### 4.4 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  $\text{void}$ , as in C, to use as the return type for a function that does not return a value. Let  $\text{Type}' = \text{Type} \cup \{\text{void}\}$ .

Symbol	Section	Meaning
$\mathbb{B}$	§4.1	$\{true, false\}$
$\mathbb{N}$	§4.1	$\{0, 1, 2, \dots\}$
ancestors	§4.1	set of ancestors of node in a tree (inclusive)
descendants	§4.1	set of descendants of node in a tree (inclusive)
length	§4.1	length of a sequence
Var	§4.2	the set of all variables
bool	§4.2	the boolean type
proc	§4.2	the process reference type
Val	§4.2	the set of all values
$Val_t$	§4.2	values of type $t$
default <sub><math>t</math></sub>	§4.2	default value of type $t$
vtype	§4.2	function $Var \rightarrow Type$ giving type of each variable
Eval	§4.2	set of all valuations on $Var$
Eval( $V$ )	§5.1	set of all valuations on variables in $V \subseteq Var$
Expr	§4.2	set of typed expressions over $Var$
etype	§4.2	$Expr \rightarrow Type$ , gives type of each expression
eval	§4.2	$Expr \times Eval \rightarrow Val$ , the evaluation function
$\mathcal{C}$	§4.2	a CIVL context
$\Sigma$	§4.3	set of all static scopes
root	§4.3	the root scope (member of $\Sigma$ )
sparent	§4.3	$\Sigma \setminus \{\text{root}\} \rightarrow \Sigma$ , parent function in static scope tree
vars	§4.3	$\Sigma \rightarrow 2^{Var}$ , specifies variables declared in scope
$\Lambda$	§4.3	a lexical scope system
void	§4.4	type used for function that does not return a value
Type'	§4.4	$Type \cup \{\text{void}\}$
$\mathcal{F}$	§4.4	set of function symbols
fscope	§4.4	$\mathcal{F} \rightarrow \Sigma \setminus \{\text{root}\}$ , gives function scope of each function
returnType	§4.4	$\mathcal{F} \rightarrow Type'$ , gives return type of each function
params	§4.4	$\mathcal{F} \rightarrow Var^*$ , formal parameter sequence for $f \in \mathcal{F}$
$f_0$	§4.4	the root function (member of $\mathcal{F}$ )
func	§4.4	$\Sigma \rightarrow \mathcal{F}$ , function to which scope belongs
Loc <sub><math>f</math></sub>	§4.7	set of locations for $f \in \mathcal{F}$
lscope <sub><math>f</math></sub>	§4.7	$Loc_f \rightarrow \Sigma$ , gives scope of each location for $f \in \mathcal{F}$
start <sub><math>f</math></sub>	§4.7	start location for $f \in \mathcal{F}$ (member of $Loc_f$ )
$T_f$	§4.7	set of guarded transitions for $f \in \mathcal{F}$

Figure 4.1: Table of Notation Used to Define CIVL Model Syntax

**Definition 4.4.0.7.** A *function prototype* for  $\Lambda$  is a tuple  $(\sigma, t, \xi)$  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 4.4.0.8.** 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 4.4.0.9.** 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  $\sigma$* .

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 4.4.0.10.** The functions *visible at scope  $\sigma$*  are those declared in  $\sigma$  or an ancestor of  $\sigma$ .

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

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

Note that the recursion in Definition 4.4.0.11 must stop as the root scope belongs to the root function and the scopes form a tree.

## 4.5 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 §5.

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  $\text{params}(f) = v_1 \dots v_n$
  - $\text{returnType}(f) = \text{vtype}(y)$
  - *meaning*: evaluate expressions  $e_1, \dots, e_n$ ; push frame on call stack and move control to guarded transition system (see §4.7) 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  $\text{params}(f) = v_1 \dots v_n$
  - *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  $\text{params}(f) = v_1 \dots v_n$
  - $\text{returnType}(f) = \text{void}$
  - $\text{vtype}(p) = \text{proc}$

- *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)

## 4.6 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.

## 4.7 Transition system representation of functions

**Definition 4.7.0.12.** 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

- $\text{Loc}$  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 statement 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 4.7.0.13.** 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 4.7.0.14.** Given a CIVL model  $M$ , let  $\text{Loc} = \bigcup_{f \in \mathcal{F}} \text{Loc}_f$ .



## Chapter 5

# CIVL Model Semantics

### 5.1 State

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

**Definition 5.1.0.15.** 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.  $\Delta$  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  $\Delta$  the structure of a rooted tree with root  $\text{droot}$ ,

4.  $\text{static}: \Delta \rightarrow \Sigma$ ,
5.  $\text{static}(\text{droot}) = \text{root}$  and  $\text{droot}$  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.  $\text{deval}$  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  $\text{Val}_{\text{proc}}$ , 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  $\text{State}$  denote the set of all states of  $M$ .

Remarks:

1. We will also refer to dynamic scopes as *dyscopes*.
2. The elements of  $\Delta$  contain no intrinsic information. Instead, all of the information concerning dyscopes is encoded in the functions that take elements of  $\Delta$  as arguments. Hence we might just as well call the elements of  $\Delta$  “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  $\delta$  is an instance of  $\sigma$  or  $\sigma$  is the static scope associated to  $\delta$ . 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`( $\delta$ ) assigns a value to each variable in the static scope associated to  $\delta$ . 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  $\delta$  and a location in the static scope associated to  $\delta$ .

**Definition 5.1.0.16.** A dyscope  $\delta \in \Delta$  is a *function node* if `static`( $\delta$ ) is the function scope of some function.

**Definition 5.1.0.17.** Given any  $\delta \in \Delta$ , `fnode`( $\delta$ )  $\in \Delta$  is defined as follows: if  $\delta$  is a function node, then `fnode`( $\delta$ ) =  $\delta$ , else `fnode`( $\delta$ ) = `fnode`(`dparent`( $\delta$ )). We call `fnode`( $\delta$ ) the *function node associated to*  $\delta$ .

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

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  $i^{\text{th}}$  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  $(\delta, l)$  be the corresponding frame; set  $\Psi(p, i) = [\delta]$ .

## 5.2 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 idea is that you have to move up the dyscope tree every time you move past a right curly brace

```

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 5.1: 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.

(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 5.1. 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  $\delta$ , and  $l'$  is in static scope  $\sigma'$ . Let  $\sigma = \text{static}(\delta)$ . Hence the current static scope is  $\sigma$  and the new static scope will be  $\sigma'$ .

First, you have to find the *least upper bound*  $\sigma_{\text{lub}}$  of  $\sigma$  and  $\sigma'$  in the static scope tree. (Hence  $\sigma_{\text{lub}}$  is a common ancestor of  $\sigma$  and  $\sigma'$ , and if  $\sigma''$  is any common ancestor of  $\sigma$  and  $\sigma'$  then  $\sigma''$  is an ancestor or equal to  $\sigma_{\text{lub}}$ .) Note that the least upper bound must exist since the function scope is a common ancestor of  $\sigma$  and  $\sigma'$ . There is a chain of static scopes from  $\sigma$  up to  $\sigma_{\text{lub}}$  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  $\delta$  and the new dynamic scope corresponding to  $\sigma'$  that will be

added to the state.

Next you add new dyscopes corresponding to the chain of static scopes leading from  $\sigma_{\text{lub}}$  down to  $\sigma'$ . 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.

### 5.3 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  $\text{jump}(s, 0, \text{start}_{f_0})$ .

### 5.4 Transitions

The transitions follow the usual “interleaving” semantics. Given a state  $\sigma$ , one defines the set of enabled transitions in  $\sigma$  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.

### 5.5 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  $P$ .

## 5.6 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 III

# Tools

## Chapter 6

# Overview of CIVL Tools

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; *replay* a trace if an error is found.

The properties checked are assertion violations, division by zero, illegal pointer accesses, etc.

The available tools and options are summarized by the `civil help` command:

Usage: `civil <command> <options> filename ...`

Commands:

- `verify` : verify program filename
- `run` : run program filename
- `help` : print this message
- `replay` : replay trace for program filename
- `parse` : show result of preprocessing and parsing filename
- `preprocess` : show result of preprocessing filename

Options:

- `-debug` or `-debug=BOOLEAN` (default: false)
  - debug mode: print very detailed information
- `-echo` or `-echo=BOOLEAN` (default: false)
  - print the command line
- `-errorBound=INTEGER` (default: 1)
  - stop after finding this many errors
- `-guided` or `-guided=BOOLEAN`
  - 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
    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
    set the random seed; applies only to run
-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
-states=STRING (default: immutable)
    state implementation: immutable, transient, persistent
-sysIncludePath=STRING
    set the system include path
-trace=STRING
    filename of trace to replay
-userIncludePath=STRING
    set the user include path
-verbose or -verbose=BOOLEAN (default: false)
    verbose mode
```



## Chapter 7

# Transitions

In CIVL, a transition stands for the execution of a number of statements of a certain process from one state, and the execution of each statement is considered as one step. A transition has the following form in the output, where `sid` and `sid'` are the identifiers for the source and the target states, `pid` is the identifier of the process that this transition belongs to, and `step 0,1 ...` are the steps contained in the transitions.

```
State sid, proc pid:
  step 0;
  step 1;
  ...
--> State sid'
```

A step of a transition has the following form in the CIVL output, where `src` and `dst` is the source and destination location of the statement executed in the step, `file` is the file that contains the source code of the statement, `location` and `text` are the location and summary of text of the statement in the source file.

```
src->dst: statement at file:location "text";
```

For example, the following is a transition from state 0 to state 1, containing 6 steps. File names are renamed with short names and the mapping of short names and the original files is presented in output as well.

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

```
State 0, proc 0:
  0->1: x = 1 at f0:3.0-9 "int x = 1";
  1->2: y = 0 at f0:4.0-9 "int y = 0";
  2->3: z = 0 at f0:5.0-9 "int z = 0";
  3->6: sum = 0 at f0:6.0-11 "int sum = 0";
  6->8: $spawn foo(1) at f0:22.4-17 "$spawn foo(1)";
  8->10: $spawn foo(2) at f0:23.4-17 "$spawn foo(2)";
--> State 1
```

To make use of this feature, one needs to specify the option “-showTransitions” when running CIVL.